Titre: Title:	Exploration architecturale de communications-sur-puce au niveau système
Auteur: Author:	Cédric Migliorini
Date:	2008
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Migliorini, C. (2008). Exploration architecturale de communications-sur-puce au niveau système [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8262/

UNIVERSITÉ DE MONTRÉAL

EXPLORATION ARCHITECTURALE DE COMMUNICATIONS-SUR-PUCE AU NIVEAU SYSTÈME

CÉDRIC MIGLIORINI
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
FÉVRIER 2008

Your file Votre référence
ISBN: 978-0-494-41569-6
Our file Notre référence
ISBN: 978-0-494-41569-6

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

EXPLORATION ARCHITECTURALE DE COMMUNICATIONS-SUR-PUCE AU NIVEAU SYSTÈME

présenté par: MIGLIORINI Cédric

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

Mme. NICOLESCU Gabriela, Doct., présidente

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. ABOULHAMID El Mostapha, Ph.D., membre

"He may be mad, but there's method in his madness. There nearly always is method in madness. It's what drives men mad, being methodical."

un Ange

REMERCIEMENTS

Tout d'abord, je remercie chaleureusement mon directeur de recherche, Professeur Guy Bois, sans qui ma maîtrise n'aurait pas eu lieu. En effet, je lui suis reconnaissant de m'avoir accueilli au sein de son laboratoire de recherche, le Circus, à la suite de deux stages réalisés en 2004 et 2005. Son expérience et ses conseils m'ont aidé à bâtir mon projet de recherche dont ce mémoire illustre l'aboutissement. De plus, j'ai apprécié la confiance qu'il a su me témoigné tout au long de ces deux années.

Une maîtrise recherche ne se déroule pas en solitaire, surtout dans un projet tel que Space. Une équipe vous entoure et chaque membre contribue à la réussite de cette maîtrise grâce à leurs connaissances, commentaires et conseils en tout genre et parfois même leur humour. Je remercie donc très sincèrement mes camarades de travail de Space Codesign : Maxime, Luc, Ahmed, Laurent, Jérôme, Sylvain, Sébastien, Benoît et Marc-André. Je ne peux oublier aussi les joyeux québécois que sont : Jean-François, Simon, Francis et Patrick pour leur complicité envers un *maudit* français, et les techniciens du GRM, Alexandre et Réjean.

Je suis très redevable au soutien sans faille de mes parents, Gyslaine et Michel, de mon frère, Yannick, et de mes grands-parents, qui même à 5000 km m'ont supporté tout au long de mes études.

Et je finirai par un énorme merci à ma copine, Agnieszka, une polonaise pas comme les autres, pour m'avoir appuyé et poussé dans mon projet de recherche depuis le début de ma maîtrise en 2006. Dziekuje bardzo !

RÉSUMÉ

Le domaine des semi-conducteurs ne cesse d'améliorer les procédés de fabrication des puces sur silicium. Aujourd'hui, les nanotechnologies offrent des densités d'intégration de transistors élevées. Cette densité accrue permet de réunir plusieurs processeurs sur une seule et même puce. Depuis quelques années, les systèmes-sur-puce multiprocesseurs sont très populaires dans des applications de haute performance telles que le multimédia, la téléphonie mobile. Les nombreux processeurs doivent être reliés par un réseau de communication efficace pour garantir le débit requis. Pour choisir la bonne architecture de communication, le concepteur se pose plusieurs questions pour satisfaire les exigences de l'application : quelle topologie choisir ? Comment la configurer ? Comment placer les composants dans cette topologie ?

Grâce à des outils ESL, une conception à plus haut niveau d'abstraction permet au concepteur d'aborder la complexité introduite par les systèmes-sur-puce multiprocesseurs et d'étudier rapidement différents aspects : communications et synchronisation inter-processeurs, topologie de mémoire, programmation parallèle, etc.
La plateforme virtuelle Space basée sur SystemC propose une méthode ESL afin de construire et d'explorer rapidement un système embarqué par assemblage de composants IP (e.g. processeurs, bus, périphériques, mémoires). Plusieurs niveaux d'abstraction permettent au concepteur d'explorer des interactions logicielles/-matérielles qu'il va associer à la meilleure architecture de communication possible.

Le projet de recherche consiste à explorer les communications-sur-puce à l'aide de la plateforme Space. Des composants à haut niveau en SystemC sont développés pour modéliser différents réseaux de communication de type multibus. Les modèles abstraits développés et améliorés touchent deux niveaux de Space. Au niveau TF, le

canal TF est capable de reproduire le fonctionnement général de bus standards du protocole CoreConnect d'IBM et AMBA de ARM, et le pont fonctionnel capable de modéliser un pont direct ou un pont adaptateur de domaines d'horloge. Au niveau BCA, les modèles précédents se raffinent en un bus OPB, bus pour périphériques du protocole CoreConnect, et en pont OPB-OPB.

Une comparaison de la simulation des architectures de communication au niveau TF et BCA montre que les modèles abstraits du niveau TF permettent une accélération de 4,3x et une estimation des communications supérieure à 85% par rapport aux modèles du niveau BCA.

Une méthode d'exploration d'architecture de communication multi-niveaux est proposée afin d'étudier les possibilités offertes par l'assemblage des précédents composants cités. Une application de décodage JPEG à laquelle on applique la méthode d'exploration démontre que la performance d'une architecture avec un simple bus partagé peut être améliorée de 14% avec un réseau multibus en plaçant judicieusement les composants sur chaque bus. Ce réseau multibus est amélioré en optimisant la configuration du protocole de chaque bus (i.e. politique d'arbitrage) et en utilisant des composants annexes tels que des liens point à point et une mémoire double. L'amélioration de performance obtenue avoisine alors les 57% pour un partitionnement tout matériel et les 82% pour un certain partitionnement matériel/logiciel. De plus, la méthode d'exploration montre aussi les risques liés à l'utilisation de ponts dans un réseau multibus comme l'interblocage et la famine.

ABSTRACT

The manufacturing processes of silicon chip are always improved by the semiconductor industry. Nowadays, the nanotechnology allows high integration density of transistors. By allowing more transistors onto a smaller area of silicon, several processors can be gathered in one chip. For a few years, multiprocessors systems-on-chip has been popular in high performance applications such as multimedia, mobile phone and gaming. The numerous processors must be linked by an efficient communication network so that the elevated bandwidth is guaranteed. The designer asks himself several questions in order to choose the right communication architecture according to the application specifications: which topology to choose? How to configure it? How to map the component onto the topology?

ESL tools enable a product design at a higher abstraction level. As a consequence, designers can face the complexity brought by multiprocessors systems-on-chip and rapidly study different aspects: interprocessors synchronization and communication, memory topology, parallel process, etc.
The virtual platform based on SystemC, Space, offers an ESL method in order to build and quickly explore an embedded system by assembling different IP blocks (e.g. processors, bus, peripherals, memory). Through several abstraction levels, the designer can test various software/hardware interactions which he will match to the best communication architecture.

This research project aims to explore on-chip communication through the platform Space. High level components are developed in SystemC in order to model different multibus communication networks. The abstract models are designed and improved for two levels of abstraction offered by Space. At the TF level, the TF channel is able to reproduce the overall mechanism of standard busses used by the IBM CoreConnect

protocol and by the ARM AMBA protocol. At this level, the functional bridge is also able to model a direct bridge or a store-and-forward bridge. These adapt to two clock domains. At the BCA level, the previous models are respectively refined into an OPB bus, peripheral bus from the CoreConnect protocol and into an OPB-OPB bridge.
A simulation of the communication architectures at the TF and BCA level is compared and shows that the TF abstract models enable an acceleration of 4,3x and a communication estimation above 85% in respect to the BCA abstract models.
A multi-levels architectural exploration method of communication was suggested so that the various assembling of the previous abstract components can be studied.
A JPEG decoder application to which the exploration method is applied reveals that the performance of a single shared bus architecture is enhanced by 14% with a multibus network by wisely mapping the components on each bus. Optimizing the protocol configuration of each bus (i.e. arbitration type) and using point-to-point links and a dual port memory improved the multibus network. Thus the performance improvement is about 57% for a pure hardware configuration and about 82% for a certain hardware/software partitioning. Moreover, the exploration method illustrates the dangers associated with the use of bridges in a multibus network such as deadlock and starvation situation.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ACRONYMES

API	**A**pplication **P**rogrammer **I**nterface
ASIP	**A**pplication **S**pecific **I**nstruction-set **P**rocessor
BRAM	**B**lock **R**andom **A**ccess **M**emory
BCA	**B**us **C**ycle **A**ccurate
CA	**C**ycle **A**ccurate
DMA	**D**irect **M**emory **A**ccess
DSP	**D**igital **S**ignal **P**rocessor
EDA	**E**lectronic **D**esign **A**utomation
ESL	**E**lectronic **S**ystem **L**evel
FIFO	**F**irst-**I**n-**F**irst-**O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HAL	**H**ardware **A**bstraction **L**ayer
HW	**H**ard**W**are
IDE	**I**ntegrated **D**evelopment **E**nvironement
IC	**I**ntegrated **C**ircuit
IP	**I**ntellectual **P**roperty
LMB	**L**ocal **M**emory **B**us
OS	**O**perating **S**ystem
MPI	**M**essage **P**assing **I**nterface
MPSoC	**M**ulti**P**rocessor **S**ystem-**O**n-**C**hip
NoC	**N**etwork-**o**n-**C**hip
NUMA	**N**on-**U**niform **M**emory **A**rchitecture
OPB	**O**n-chip **P**eripheral **B**us
PIC	**P**in **I**nterrupt **C**ontroller
PLB	**P**rocessor **L**ocal **B**us
RAM	**R**andom **A**ccess **M**emory

SW	**S**oft**W**are
SMP	**S**ymmetric **M**ulti**P**rocessing
SoC	**S**ystem-**O**n-**C**hip
RTL	**R**egister **T**ransfer **L**evel
RTOS	**R**eal **T**ime **O**perating **S**ystem
TF	**T**imed **F**unctionnal
TLM	**T**ransactional **L**evel **M**odel
UMA	**U**niform **M**emory **A**rchitecture
UTF	**U**ntimed **T**imed **F**unctionnal
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage

LISTE DES TABLEAUX

LISTE DES ANNEXES

INTRODUCTION

Un **système embarqué** est un système informatique et électronique autonome dans lequel le matériel et le logiciel sont très fortement couplés. Il est conçu pour réaliser une fonction spécifique. Les systèmes embarqués se trouvent à la maison (télévision numérique, lecteur DVD, système de son), au travail (assistant numérique personnel, routeur et commutateur dans les réseaux), dans les loisirs (téléphone cellulaire, console de jeux vidéos, lectuer MP3) et les transports (GPS, aide d'assistance au freinage). Ils intègrent le plus souvent un ou plusieurs processeurs, des périphériques d'entrée/sortie et de la mémoire sur une même puce. Nous parlons alors de système-sur-puce (SoC). Les processeurs embarqués sont des processeurs à usage spécial conçus pour une classe d'applications spécifiques.

La conception des systèmes embarqués passent par le développement conjoint du matériel et du logiciel : c'est le *codesign*. Dans ce modèle, une communication constante est nécessaire entre les deux équipes de conception. Le résultat est que chacune des équipes peut bénéficier du travail de l'autre. La composante logicielle peut profiter des caractéristiques avancées du matériel pour gagner en performance. La composante matérielle peut simplifier la conception d'un module si la fonctionnalité peut être mise en logiciel pour réduire le coût et la complexité du matériel.

Jusqu'au début des années 90, les systèmes embarqués étaient généralement des systèmes simples avec une longue vie sur le marché. Depuis les dix dernières années, l'industrie de l'embarqué connaît des transformations importantes conduisant à de nouvelles contraintes. Ils sont conçus en fonction des exigences de performance, de taille, de consommation de puissance et de prix.

En 1965, Gordon Moore, co-fondateur d'Intel, prédisait que le nombre de transistors

par circuit intégré doublerait tous les 24 mois. Des estimations plus affinées dans les années suivantes donnèrent un intervalle de 18 mois.

La loi de Moore est devenue un objectif de performance pour l'ensemble de l'industrie de la microélectronique. Pendant 25 ans, les procédés de fabrication des circuits intégrés (IC) ont toujours rendu possible l'intégration des transistors. Mais les outils et les méthodes de conception n'arrivaient pas à suivre la cadence [48]. Ce phénomène est appelé **l'écart de productivité**. Les concepteurs matériels d'IC sont donc montés vers des niveaux d'abstraction plus élevés dans les outils pour répondre à la complexité grandissante de conception tout en respectant les temps de mise sur le marché. Mais la loi de Moore continue à faire avancer les procédés de fabrication (2003 : 130 nm, 2005 : 90 nm, 2007 : 65 nm)[17]. Cet écart de productivité risque de durer encore une décennie (figure 1) jusqu'à ce que la loi de Moore ne soit plus valable car les limites atomiques du silicum auront été atteintes [17, 48, 66].

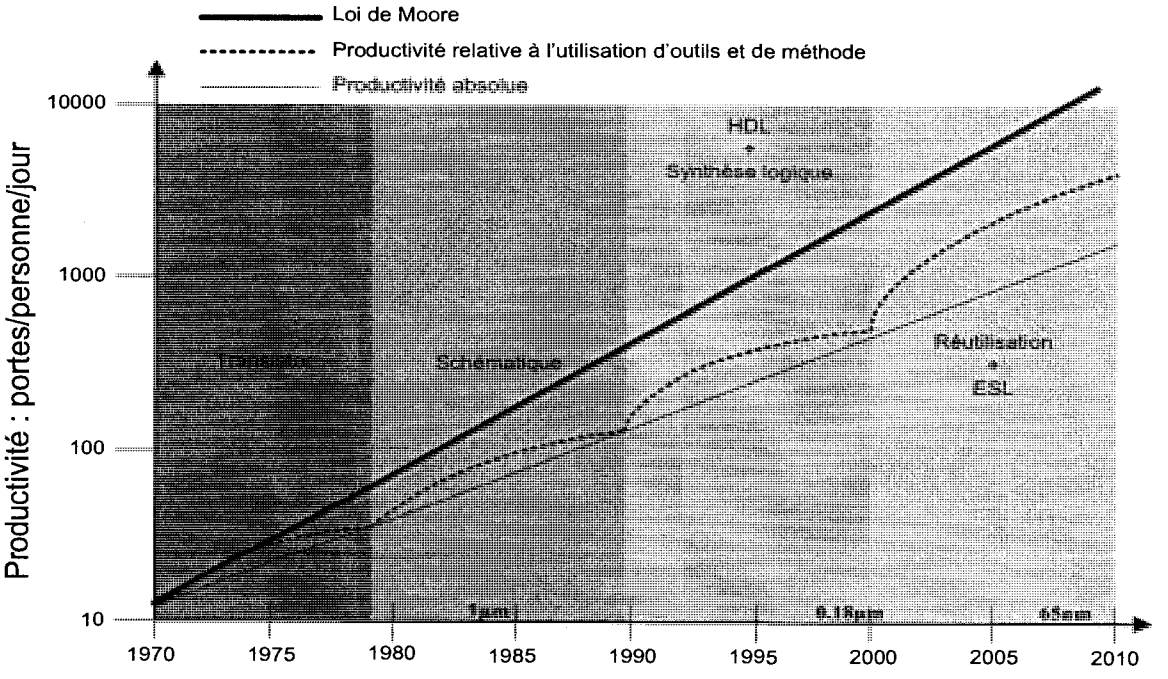

Figure 1 Ecart de Productivité

Problématique

La conception d'un système-sur-puce doit donc relever des défis dont certains sont induits par la loi de Moore.

1. La **mise sur le marché** (*time-to-market*). Le processus de conception d'un produit doit être le plus court possible pour répondre à la demande.

2. Le "triangle technologique" **performance/énergie/coût**. Le SoC doit garantir une performance optimale tout en minimisant la dissipation de puissance dégagée par les différents composants. La gestion de l'énergie devient cruciale pour les systèmes embarqués mobile fonctionnant sur batterie.

3. La **densité d'intégration** augmente la complexité des SoCs mais réduit le coût global en combinant plusieurs fonctionnalités sur une surface plus petite.

4. Le **temps**. Certains systèmes doivent garantir un débit rapide et exécuter des tâches dans un laps de temps précis.

La venue de **systèmes-sur-puce multiprocesseurs** (*MPSoC*) depuis quelques années permet notamment de satisfaire le triangle technologique. Les millions de transistors par puce apportés par la loi de Moore au $21^{\text{ème}}$ siècle permettent l'utilisation de plusieurs processeurs. L'association de processeurs de nature homogène ou hétérogène vise à diminuer la fréquence du système et donc à baisser la puissance dissipée [48]. Les MPSoCs offrent un avenir prometteur au **parallélisme**. Ils continuent d'exploiter le parallélisme inhérent au matériel et l'introduisent dans le logiciel, ce qui oblige les programmeurs à penser parallèle au lieu de séquentiel.

Les MPSoCs sont très employés dans le domaine de la réseautique, de la téléphonie mobile, du multimédia et de l'automobile. Le point commun de ces domaines est la quantité importante de données générées par l'application. Pour que les processeurs et les autres composants puissent s'échanger ces données, ils ont besoin

d'un **réseau de communication** performant. Ce réseau doit faire face à la complexité croissante des systèmes qui engendrent un trafic accru et doit répondre aux exigences de performance et d'énergie imposées par la nanotechnologie [47]. Diverses architectures de communication permettent de satisfaire les contraintes évoquées. Du simple bus partagé au lien point à point en passant par un réseau multibus et des réseaux de communication plus complexes, les topologies ne manquent pas [44][10]. Le concepteur se retrouve donc devant plusieurs alternatives et le choix et l'optimisation de la topologie devient vite un casse-tête.

Les outils ESL (*Electronic System Level*) au **niveau système** permettent d'explorer différentes architectures de communication pour relier les différents composants dans un MPSoC. Grâce à plusieurs niveaux d'abstraction et à des modèles abstraits des blocs IP matériels, une application entière (logiciel et matériel) peut être simulée et vérifiée. Les modèles abstraits des blocs IP sont écrits dans un langage de haut niveau comme SystemC [59]. Les concepteurs pourront plus tôt dans le cycle de conception évaluer les interactions matérielles/logicielles à travers le réseau de communication, trouver le meilleur compromis et réduire les risques de conception. En personnalisant le réseau de communication, en fonction des spécifications de l'application et du domaine d'application, plusieurs aspects sont à considérer :

1. sélection de la topologie du réseau de communication.

2. placement des divers composants dans le réseau de communication.

3. sélection des bons paramètres du protocole du réseau de communication.

La plateforme de conception SPACE [16] basée sur SystemC répond aux critères précédents. Elle permet de construire un système à partir d'une librairie de composants IP à haut niveau. À travers plusieurs niveaux d'abstraction, le système est validé fonctionnellement et raffiné vers une implémentation plus détaillée en termes de temps. Une fois le système conçu à haut niveau, il peut être vérifié au niveau

RTL en ciblant la technologie FPGA. SPACE cible particulièrement les FPGAs de la société Xilinx.

Objectifs

L'objectif principal de ce projet est d'explorer différentes architectures de communication à l'aide de la plateforme SPACE. Les architectures visées sont le bus simple partagé, les multibus reliés par des ponts et les liens point à point. Plus particulièrement, les modèles des architectures de communication de haut niveau reproduisent le fonctionnement du protocole CoreConnect d'IBM constitué de trois bus et des liens point à point développés par Xilinx. CoreConnect et les liens point à point sont utilisés dans le FPGA Virtex II Pro de Xilinx.

À travers les trois niveaux d'abstraction (UTF, TF et BCA) offerts par SPACE, deux buts plus spécifiques sont montrés. Le premier démontre que l'approche de haut vers le bas de SPACE permet d'arriver à un système avec une architecture de communication répondant aux contraintes de l'application choisie. Le deuxième illustre que les outils d'analyse graphique et textuelle de SPACE permettent de personnaliser l'architecture de communication à partir de métriques quantitatives et qualitatives pour optimiser la performance globale du système. Les deux buts visent à proposer une méthodologie d'exploration des communications-sur-puce multi-niveaux d'abstraction.

Méthodologie

Dans un premier temps, les caractéristiques communes à des bus standards seront analysées. Les bus standards CoreConnect d'IBM [29] et AMBA de ARM [5] seront utilisés pour trouver ces caractéristiques. Ces derniers permettront de développer un modèle transactionnel de haut niveau des bus écrit en SystemC et hautement configurable pour reproduire la plupart des fonctionnalités.

Dans un système multibus, les bus sont reliés par des ponts. De la même manière que

pour les bus, les caractéristiques communes des ponts seront extraites de l'analyse des ponts utilisés dans le protocole CoreConnect et AMBA. Ceci permettra de développer deux modèles transactionnels en SytemC respectivement pour les niveaux TF et BCA. Tous les modèles transactionnels seront testés et optimisés par des bancs de tests.

Par la suite, à partir d'un banc de test mettant en œuvre les communications propres à SPACE, les modèles transactionnels développés pour le niveau TF et BCA seront comparés à travers une configuration simple bus et multibus avec un pont.

Enfin, une application de décompression d'images JPEG sera utilisée au niveau TF et BCA pour concevoir un système performant basé sur une architecture de communication optimisée à l'aide de bus. Au niveau BCA, le bus OPB du protocole CoreConnect d'IBM est utilisé. Cette application permettra de mettre en lumière une méthodologie d'exploration itérative avec SPACE pour choisir une architecture multibus, placer les composants sur cette architecture et personnaliser le protocole de l'architecture. Les pièges liés à l'utilisation d'un pont seront aussi étudiés et l'utilisation d'une mémoire double port et de liens point à point montrera comment améliorer le système multibus.

Contributions

Ce travail a tout d'abord permis de développer des modèles de haut niveau que sont le canal transactionnel TF et le pont fonctionnel au niveau TF, le pont OPB-OPB et quelques fonctionnalités supplémentaires du bus OPB au niveau BCA. Les modèles abstraits au niveau TF permettent d'obtenir un gain de performance sur la simulation réalisée au niveau BCA avec une bonne estimation des communications.

Une méthodologie d'exploration des communications-sur-puce a été développée montrant la facilité d'exploration architecturale à travers les différents niveaux d'abstractions de SPACE. De plus, grâce aux outils d'analyse graphique et textuelle

de SPACE, les problèmes liés à l'utilisation d'un réseau multibus sont facilement identifiables et solvables à l'aide de composants annexes comme une mémoire double port et des liens point à point.

Organisation du mémoire

Ce mémoire se compose de quatre chapitres. Le premier chapitre présente les tenants et les aboutissants de la conception des systèmes-sur-puce multiprocesseurs. Le deuxième chapitre présente plus particulièrement les types de communications-sur-puce et les techniques d'analyse et d'exploration architecturale de ces communications. Le troisième chapitre décrit brièvement les différents niveaux de la plateforme SPACE et présente les composants abstraits du niveau TF et BCA permettant de construire et d'améliorer un réseau multibus. Il décrit aussi la méthode développée pour mener une exploration architecturale des architectures de communication multibus à travers les niveaux de SPACE. Enfin, le dernier chapitre présente les résultats comparant la vitesse et la précision des simulations entre le niveau TF et BCA ainsi que les performances obtenues lors de l'exploration architecturale des communications à travers les niveaux d'abstractions de SPACE avec la méthode proposée.

CHAPITRE 1

SYSTÈME-SUR-PUCE MULTIPROCESSEUR

Une architecture de type système-sur-puce (*SoC*) intègre plusieurs composants hétérogènes sur une seule puce. Ce type de système permet de construire des applications complexes et spécialisées. Depuis quelques années, des applications dans le domaine des télécommunications, du multimédia, du réseau et bien d'autres favorisent le développement de plateforme à très haute performance. Pour répondre à ce besoin de performance, le concept de système-sur-puce multiprocesseur (*MPSoC*) a vu le jour. Ainsi, ce chapitre présente les enjeux liés à l'émergence des MPSoCs.

Le concept de SoC a émergé au début des années 2000. En général, les SoCs sont composés : d'un ou plusieurs microcontrôleurs, processeurs et DSPs, de plusieurs types de mémoire, d'une source pour l'horloge du système, d'interfaces d'entrées/sorties, d'autres périphériques, et parfois de la logique spécifique à l'application. Tous ces blocs sont reliés par un réseau d'interconnexion tels que des bus standard ou par un réseau-sur-puce. Aujourd'hui, le terme de *puce* est associé aux éléments que nous trouvons dans un SoC. Ainsi, nous parlerons de communication-sur-puce (*on-chip communication)* tels que les bus- et réseaux-sur-puce (*bus-/network-on-chip*) et de mémoire-sur-puce (*on-chip memory*).

1.1 Le besoin grandissant

Les systèmes-sur-puce multiprocesseurs (*Multiprocessor System-On-chip, MPSoC*) sont les derniers nés de la technologie VLSI (*Very-Large- Scale Integration*). Ils sont apparus il y a quelques années et percent de plus en plus le marché. Les MPSoCs

trouvent leur sens dans le marché de l'électronique de masse dans lequel les produits doivent être performants, consommés peu d'énergie et avoir un prix abordable. Parmi les domaines répondant aux critères précédents, il y a les télécommunications, le réseau, la téléphonie mobile, la télévision numérique et les jeux vidéo. À l'inverse de l'ordinateur à usage général où la course à la fréquence entre Intel et AMD a atteint les 4 GHz, les MPSoCs s'appuient sur l'utilisation de plusieurs processeurs pour minimiser la fréquence de fonctionnement et donc la consommation d'énergie. Ces processeurs peuvent être configurables, à jeu d'instructions fixes ou à jeu d'instructions extensibles [52].

Les nouvelles technologies requièrent du **parallélisme** car elles intègrent plusieurs fonctionnalités dans un même appareil (audio/vidéo, sans fil, sécurité, contrôle). Ces applications impliquent que de multiples algorithmes soient utilisés en concurrence. Il en découle de multiples instructions par cycle et de nombreuses données à traiter.

De nos jours, un téléphone cellulaire comporte de quatre à huit processeurs incluant un ou plusieurs processeurs RISC (*Reduced Instruction-Set-Simulator*) pour les interfaces graphiques et d'autres fonctions de contrôles, un DSP pour l'encodage et le décodage de la voix, des processeurs pour la caméra numérique, la vidéo et l'écoute de la musique. Cet exemple montre un des aspects importants des MPSoCs, **l'hétérogénéité.** En effet, multiples processeurs pour multiples fonctionnalités, chaque fonction est réalisée par un processeur spécialisé ou le mieux adapté pour obtenir la meilleure performance.

La recherche de **performance** est un enjeu puisque les applications pour les MPSoCs sont soumises à des contraintes de temps réel. Les systèmes sont mobiles dans un environnement qui contient un nombre incalculable d'informations qui se propagent dans les airs sous forme d'ondes (lumineuse, radio, etc). Certaines tâches du système doivent s'exécuter dans un laps de temps précis.

Que ce soit des appareils avec batterie comme le téléphone cellulaire ou sans batterie comme la télévision numérique, la soif **d'énergie** est à contrôler. Il faut le plus possible minimiser la consommation de puissance afin par exemple d'allonger la durée de vie de la batterie. Ce contrôle permet notamment de réduire les **coûts**.

Le fameux triangle **performance/énergie/coût** est au cœur de la conception des MPSoCs. Aussi bien sur le plan matériel que logiciel, ce triangle apporte de nouveaux défis à surmonter pour les ingénieurs systèmes, logiciels et matériels, vérification et intégration. Cela passe par l'amélioration des outils et les méthodes existantes pour faire face à la complexité grandissante. Deux pistes s'offrent aux concepteurs pour relever ces défis [19].

La première solution passe par la **réutilisation** de blocs IP. Les concepteurs ont accès à une librairie de composants ou de blocs déjà conçus et testés dans laquelle ils peuvent se servir pour rapidement construire leur système en assemblant les différents IPs.

La deuxième solution passe par l'adoption de la conception au **niveau système**, que l'on nomme *Electronic System Level* (*ESL*). L'ESL implique que les librairies d'IPs offriront des modèles abstraits des composants décrits dans un langage de haut niveau comme SystemC [59]. Ainsi, les concepteurs pourront plus tôt dans le cycle de conception évaluer les interactions matérielles/logicielles, trouver le meilleur compromis et réduire les risques de conception. Grâce à plusieurs niveaux d'abstraction et à un modèle comportemental des IPs matériels, une application entière (logiciel et matériel) peut être simulée et vérifiée avant d'arriver au flot de conception matériel traditionnel.

1.1.1 L'avenir est dans l'ESL

La conception au niveau ESL (*Electronic System Design*) peut être vue comme une méthodologie générale qui inclut la conception et la simulation, la synthèse comportementale, et la vérification de systèmes-sur-puce à haut niveau d'abstraction. Elle fournit des outils qui utilisent diverses méthodologies pour décrire et analyser les SoCs sur différents niveaux d'abstractions [33]. Il n'est pas toujours évident de trouver une définition claire car l'ESL prend plusieurs formes. En effet, dans [53], l'auteur stipule que les outils, les modèles et les méthodologies viennent de différentes sources. Cela passe par des chercheurs universitaires, des fournisseurs d'IPs, des fournisseurs d'outils commerciaux pour l'ESL, des entreprises classiques de l'EDA (*Electronic Design Automation*).
Malgré une diversité apparente et la difficulté d'établir un flot ESL standard, nous pouvons tout de même caractériser l'ESL.

1.1.1.1 Conception et simulation au niveau système

Cette partie couvre deux aspects [33].
Le premier est la conception **algorithmique** qui associe des données et des scénarios réels à un algorithme afin de mieux vérifier et valider les résultats finaux et le comportement. L'outil Matlab associé à Simulink est depuis longtemps utilisé.

Le deuxième est l'utilisation de **modèle de haut niveau**, appelé aussi **prototype virtuel**, pour l'exploration architecturale et la vérification. Le prototypage virtuel permet de développer le logiciel plus efficacement et de valider les interactions logicielles/matérielles. Dans les deux prochaines années, son utilisation atteindra 44% contre 28% en 2007 grâce aux outils ESL commerciaux [58]. Un prototype virtuel est donc un modèle architectural caractérisée par la création d'une plateforme exécutable

sur laquelle le vrai code logiciel compilé et lié s'exécute. Elle inclut un ou plusieurs processeurs, des bus, de la mémoire et des périphériques dont les modèles haut niveau sont précis au cycle (*cycle-accurate*). Ceci permet de recréer le comportement réel du système de façon précise. Un prototype virtuel offre un compromis entre la vitesse de simulation et la précision des résultats. Ainsi, dans un flot de conception complet de système (Figure 1.1), l'utilisation d'un prototype virtuel permet de garantir une meilleure cohérence entre le choix de l'architecture et son implémentation.

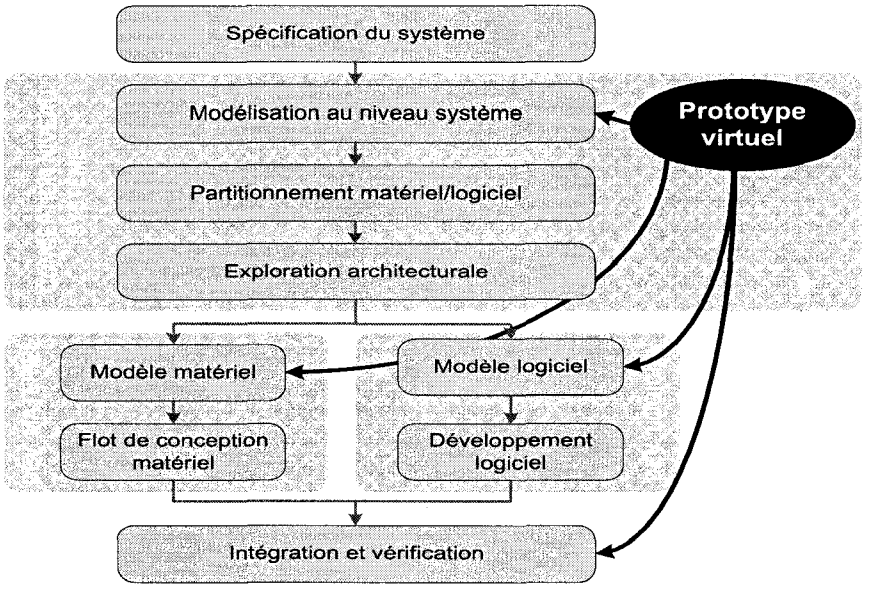

Figure 1.1 Prototype virtuel

La modélisation au niveau transactionnel (*Transaction-Level Modeling, TLM*) est un concept de plus en plus populaire dans la conception au niveau système. Elle repose sur l'utilisation de plusieurs niveaux d'abstractions et sur le raffinement progressif et indépendant de la communication et des fonctionnalités du système. Actuellement, l'*Open SystemC Group Initiative* (*OSCI*) définit un ensemble de terminologie et propose un standard pour le TLM [73]. Le TLM s'appuie sur la notion de **transaction**. Une transaction est un échange d'information entre deux composants. En fonction du niveau d'abstraction, la transaction peut désigner tous les phases, une phase particulière ou un signal particulier utilisé dans une

phase de l'échange de l'information. La communication entre deux composants est modélisée par un canal abstrait et les transactions s'échangent en appelant des fonctions à travers une interface fournie par le canal. Par exemple, pour un envoi de donnée sur un canal selon le niveau d'abstraction, une transaction pourrait définir toutes les phases de l'envoi (i.e. arbitrage, décodage d'adresse, transfert, acquittement) ou alors chaque phase de l'envoi implique une transaction. La notion de transaction est intimement liée à celle d'événement. La gestion d'une ou plusieurs transactions entraine la gestion d'un ou plusieurs évènements. Au sein d'un même composant, les détails de la communication sont séparés des détails de calcul. Les détails non nécessaires de la communication et des calculs sont cachés et pourront apparaître plus tard en descendant dans les niveaux [12]. Ainsi, grâce au TLM, des modèles de bus et de réseaux-sur-puce peuvent être simulés plus rapidement que des modèles RTL tout en ayant une grande précision lors de la simulation [42]. Les simulations TLM permettent d'extraire des métriques telles que la puissance, la performance, la surface et d'évaluer les interactions matériel/logiciel en explorant rapidement plusieurs alternatives d'implémentation. De ce fait, une tendance à un flot de conception TLM vers RTL (*Register Transfer Level*) émerge afin d'assurer la transition entre la conception système et la conception traditionnel au niveau RTL [68].

1.1.1.2 Les niveaux d'abstraction pour appréhender la complexité

L'ESL couvre plusieurs niveaux d'abstractions qui servent à raffiner les modèles des composants à haut niveau. Ces niveaux d'abstractions sont au nombre de quatre.

Le niveau **UTF** (*UnTimed Functional*) ne contient pas de notion de temps. Il permet de tester la fonctionnalité sans préjuger d'un quelconque partitionnement logiciel/matériel. Les communications s'effectuent en un temps zéro.

Le niveau **TF** (*Timed Functional*) fait apparaître la notion de temps mais pas à travers une horloge. Les opérations de calcul des composants et les communications sont associées à un temps représentant leur exécution. La simulation donne ainsi des premières estimations temporelles. Le partitionnement logiciel/matériel est rendu possible.

Le niveau **BCA** (*Bus Cycle Accurate*) permet de modéliser les communications entre les composants à travers un réseau de communication tel qu'un bus ou un réseau-sur-puce. Ce niveau est dit « précis au cycle » car les communications sont découpées en phase permettant de savoir ce qui passe cycle par cycle. Une notion d'horloge peut apparaître générant des évènements supplémentaires à gérer. Ce niveau est utile pour la phase de vérification au niveau système car la simulation en termes de cycle d'exécution se rapproche du système réel.

Le niveau **CA** (*Cycle Accurate*) modélise en détails un système en incluant le protocole complet de communication avec tous les signaux d'entrée/sortie. Une ou plusieurs horloges servent à synchroniser tous les composants du système.

Le TLM couvrent les trois premiers niveaux qui permettent un raffinement selon deux axes : la granularité des données (paquet de l'application, paquet sur le bus, taille du bus) et la précision sur le temps (pas de temps, temps approximé, temps précis au cycle). Les appellations ci-dessus peuvent changer selon les organismes [27][71][78]. La figure 1.2 montre une possible correspondance et les principales caractéristiques :

À partir de ce schéma, nous pouvons dégager deux types d'approches dans l'ESL. L'approche du haut vers le bas (*top-down*) où nous partons d'une modélisation à haut niveau en ajoutant au fur et à mesure les détails pour arriver à une implémentation physique. Cette approche est souvent adaptée à une vue utilisateur/concepteur de système. Et l'approche du haut vers le bas (*bottom-up*) dans laquelle nous partons de la structure et de l'implémentation du système pour remonter en abstrayant les

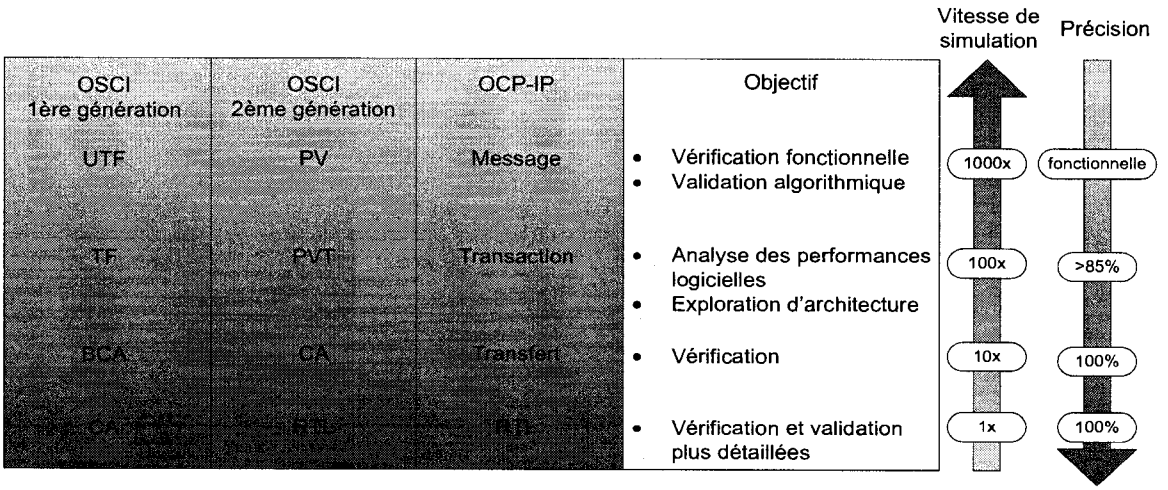

Figure 1.2 Niveaux d'absractions

détails de plus en plus. Cette approche est plutôt utilisée par des développeurs d'IPs à haut niveau. Ces IPs haut niveau serviront justement à un concepteur de système dans l'autre approche.

1.2 Le logiciel

Dans la conception des MPSoC, le logiciel représente une part importante voir critique. Avant, le concepteur de puce pouvait trouver des solutions purement matériel pour répondre aux spécifications du système. Maintenant, puisque les fonctionnalités peuvent se partitionner en matériel ou en logiciel, les concepteurs matériels doivent composer avec le logiciel dès le début. Ils doivent comprendre les caractéristiques de l'application logicielle qui affectent les contraintes temps réel et la consommation de puissance. Pour comprendre les défis logiciels, le découpage en trois points de vue proposé par Wolf et Jerraya [37] sera retenu : 1) la vue programmeur; 2) la vue architecture logicielle et réutilisation de la méthode de conception; 3) la vue optimisation.

1.2.1 La vue programmeur

Pour exploiter le parallélisme offert par les MPSoCs, une **programmation parallèle** est nécessaire pour associer l'application logicielle à la meilleure implémentation. Il existe des **librairies standards de programmation parallèle** appelées API (*Application Programmable Interface*) permettant d'exploiter au maximum l'architecture multiprocesseur offerte. Par exemple, le standard OpenMP est adapté pour la programmation à mémoire partagée, le standard MPI (M*essage-Passing Interface*) offre une programmation par passage de messages. Les modèles de programmation doivent offrir les mécanismes adéquats pour éviter les problèmes liés à la concurrence tels que les interblocages, les famines de données, les incohérences de données, les concurrences critiques (*race conditions*).

1.2.2 La vue architecture logicielle et la réutilisation de la méthode de conception

L'architecture logicielle permet à l'application logicielle de s'exécuter sur l'architecture MPSoC. Elle comprend l'intergiciel, le système d'exploitation (très souvent temps réel, *RTOS*) et la couche d'abstraction matérielle (*Hardware Abstraction Level, HAL*). Le HAL est un composant logiciel qui dépend directement du processeur (e.g. pilotes pour le bus, unité de gestion de la mémoire, routine de service d'interruption). L'architecture logicielle permet la communication entre les tâches logicielles composant l'application, l'ordonnancement de ces tâches et la gestion des événements extérieurs (e.g. interruptions). Ainsi, dans un contexte MPSoC, le RTOS et l'intergiciel doit offrir au minimum un service d'ordonnancement, de gestion de mémoire, de gestion de communication et de synchronisation. Certains de ces services doivent s'exécuter le plus efficacement possible pour respecter les contraintes de temps et de puissance imposées par l'application [80]. Comme la conception MPSoC est régie

par des contraintes de performance et de coût, l'architecture logicielle doit pouvoir être configurable pour n'utiliser que les fonctions nécessaires et ainsi minimiser la surcharge [37].

1.2.3 La vue optimisation

Le code logiciel dans les MPSoCs doit être optimisé en termes de taille de code, de temps d'exécution et de consommation d'énergie pour répondre aux contraintes de surface, de puissance et de temps. La performance logicielle dépend fortement du **type de processeur** (DSP, ASIP, processeur configurable, etc.) et de la **hiérarchie de mémoire** (partagée ou distribuée)[37].

1.3 Le matériel

Dans la section précédente, les défis logiciels pour une conception multiprocesseur ont été présentés mais qu'en ait-il des défis matériels ? Les questions architecturales suivantes établies dans [37][52] aident à comprendre ces défis :

- Quel type de processeur utiliser ? Combien de processeurs faut-il et comment les configurer en fonctions de l'application ? Faut-il une architecture homogène ou hétérogène ?
- Quelle hiérarchie de mémoire ? Où placer les mémoires et combien de mémoire allouée aux différentes tâches ?
- Quel type d'interconnexion et topologie faut-il utiliser (communications point à point, bus standard, NoC ou un mélange) ?

1.3.1 Les processeurs

Selon une étude de mars 2006 [24] sur les types de processeurs utilisés dans les systèmes embarqués, les processeurs 32 bits représentent 55% du marché suivi par les processeurs 16 bits à 18% et les 8 bits à 16%. Cette étude montre aussi que 44% des systèmes fonctionnent entre 10 et 99 MHz ce qui correspond à une recherche de consommation de puissance contrôlée. Seulement 28% des systèmes sont conçus pour travailler entre 100 et 500 MHz et 13% entre 500 et 1GHz. Une autre étude réalisée dans le cadre de Linux pour l'embarqué en mai 2006 [49] montre que ARM était présent dans 30% des projets en 2004 et 2005 alors que Intel x86 était présent dans 28%. Suivent MIPS et PowerPC avec respectivement 10% et 14% des projets.

Le choix du processeur dans un système MPSoC va dépendre de multiples facteurs tels que la performance, le coût et la consommation de puissance de la puce à concevoir, les outils de développement logiciel et matériel disponible pour le processeur, les systèmes d'exploitation qu'il supporte, la possibilité de le configurer et d'étendre son jeu d'instruction, les intergiciels et les pilotes disponibles pour interagir avec les périphériques. Pour le système MPSoC, certains facteurs seront plus importants que d'autres en fonction des spécifications [24].

Les processeurs pour l'embarqué peuvent être classés selon le graphe 1.3[21].

Les processeurs d'usage générale offre un haut degré de programmation via une architecture pour un jeu d'instruction généralisé. Il supporte des langages tels que le C/C++. Il exploite surtout le parallélisme d'instructions en utilisant la technique de pipeline. Le pipeline augmente la performance d'exécution en traitant simultanément plusieurs instructions grâce à de multiples étages [36]. Ces processeurs sont basés sur une architecture CISC ou RISC. Le PowerPC d'IBM [32], le MIPS [55], le ARM [6] sont des exemples de processeur d'usage général. Un ASIP (*Application*

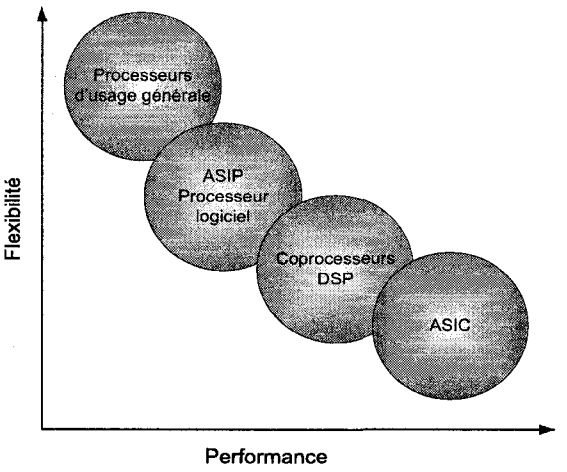

Figure 1.3 Classification des processeurs embarqués

Specific Instruction Processor) est un processeur configurable qui permet d'ajouter ou d'enlever des composants pour mieux s'ajuster à une application. Le Xtensa de Tensilica est un exemple. Un processeur logiciel (*soft processor* ou *softcore*) est un processeur configurable selon les besoins. Il est employé dans la technologie FPGA. Il est décrit logiquement en VHDL ou Verilog et est synthétisé avec le reste d'une plateforme matérielle. Le Microblaze de Xilinx [86] ou le Nios de Altera [3] en sont des illustrations. Les processeurs logiciels s'opposent aux processeurs matériels (*hard processor* ou *hardcore*) qui sont physiquement câblé dans le FPGA et sont optimisés pour le minutage (*timing*) et la consommation de puissance. Les exemples précédents de processeurs d'usage général sont considérés comme des processeurs matériels car ils sont présents dans les FPGA de Xilinx, de QuickLogic et d'Altera respectivement [87]. Un DSP (*Digital Signal Processing*) est un processeur spécialisé pour des calculs intenses comme le traitement de signaux analogiques. Ils sont souvent d'architecture CISC à point flottant ou à point fixe. Les processeurs TMS320Cxx de Texas Instruments [35] sont des exemples de DSP.

Les architectures **homogènes** de processeurs permettent via une mémoire principale partagée une programmation facile pour traiter de grande quantité de données.

Elles sont donc idéales pour des applications orientées données. Si le système d'exploitation le supporte, l'homogénéité des processeurs permet de bouger les tâches entre processeurs pour équilibrer la charge du système. Alors que les architectures **hétérogènes** de processeurs permettent d'optimiser le système en enlevant et ajoutant des fonctions. Ceci permet de réduire la consommation d'énergie du système pour une catégorie d'applications [80].

1.3.2 La mémoire

Le système de mémoire influence énormément les contraintes de temps réel et d'énergie d'un MPSoC. Elle agit sur la consommation d'énergie à cause des accès aux données (lectures/écritures), à la cohérence des données requises dans le cas de données partagées et du stockage des données [38].

L'architecture à mémoire partagée facilite la programmation car un seul espace d'adressage est défini pour tous les processeurs. Deux processeurs peuvent communiquer implicitement par des variables partagées ou explicitement par des mécanismes de blocage/déblocage (sémaphore, boîtes aux lettres, mutex). Il existe deux types de mémoire partagée montrés dans la figure 1.4: accès à la mémoire uniforme (*Uniform Memory Access, UMA*) et accès à la mémoire non uniforme (*Non-Uniform Memory Access, NUMA*). Dans le premier cas, tous les accès mémoires ont la même latence pour les processeurs car ces derniers sont connectés ensemble à un bus ou un réseau-sur-puce. Les mémoires caches et locales peuvent aider à réduire la contention sur le bus. Dans le deuxième cas, la mémoire partagée distante à une plus grande latence que les accès aux mémoires locales. L'espace d'adressage est visible par tous les processeurs. Ce système offre une plus grande évolutivité mais est plus difficile à programmer.

D'après [80], la tendance d'architecture de mémoire est une mémoire partagé NUMA

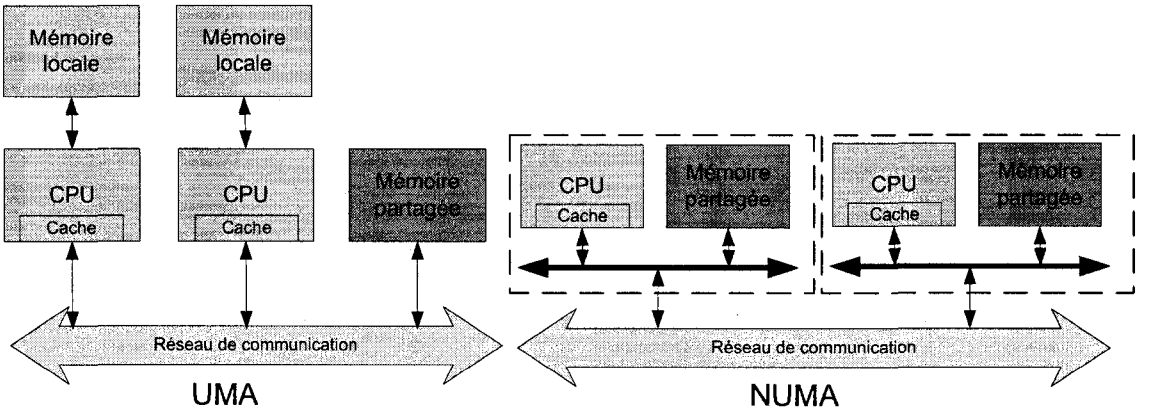

Figure 1.4 Architecture de mémoires

car celle-ci permet de mieux appréhender les parties critiques en terme de temps d'exécution. La construction d'une mémoire plus spécialisée permet de mieux prédire le comportement du système pour observer les effets lors de l'exécution et de minimiser la consommation d'énergie. De plus, un certain nombre de techniques existent pour construire une hiérarchie de mémoire correcte et optimiser l'application logicielle pour cette hiérarchie.

1.3.3 Les communications-sur-puce

Les communications-sur-puce jouent un rôle de plus en plus important dans les MPSoCs [13]. Comme la technologie de fabrication atteint le nanomètre, le délai de propagation dans les fils, la dissipation de puissance ont la plus haute importance. Plus de composants sur une puce implique une meilleure architecture de communication capable de maintenir et d'augmenter la bande passante requise pour le système. Ainsi, les architectures basées sur les bus sont les premiers à voir le jour dans les SoCs. En se servant de l'expérience acquise dans l'informatique de bureau, les bus se sont imposés comme la solution logique. Les bus permettent de relier plusieurs composants maîtres (processeurs, coprocesseurs, DMA) à des composants

esclaves (périphériques en tout genre). Comme l'accès au bus est partagé, un arbitre est souvent utilisé. Pour faire face à la complexité des MPSoCs, diverses options à des bus standards comme CoreConnect d'IBM ou Amba de ARM ont été ajoutées : transactions scindées (*split-transaction*), mode rafale, techniques de pipeline du bus de données, plusieurs domaines d'horloge par l'utilisation de bus hiérarchiques. D'autres types de communication telles que les liens point à point permettant de relier directement deux composants sont venus améliorer les performances du système. Puis depuis quelques années, face à la diversité des types de trafic générés par les applications, les architectures réseaux connus dans le monde d'Internet arrivent dans le monde des MPSoCs. Ainsi, sont nés les réseaux-sur-puce [8] qui à l'aide de commutateur et d'algorithmes de routage relient les différents composants. Toutes ces nouvelles architectures sont suivies par diverses méthodes [40] de conception pour en tirer profit.

Cette section fait l'objet d'un développement plus approfondie dans le prochain chapitre.

1.4 Méthodologies et plateformes de conception

Classifier les méthodes et les plateformes de conception MPSoC pour l'ESL de façon claire et précise n'est pas chose facile tellement la diversité des définitions d'ESL est grande. Ainsi, l'article [20] servira de guide pour cette classification. Cette dernière se base sur une structure en Y où les trois branches sont la *fonctionnalité* (représentation fonctionnelle du système indépendante de toute architecture), la *plateforme* (les modules, i.e. processeurs, mémoires, etc, pour implémenter la description fonctionnelle sont considérés) et l'*association* (la fonctionnalité a été assignée à un ensemble de modules interconnectés). Ici sont présentées des plateformes industrielles et académiques incluant l'aspect MPSoC dans le flot de conception.

1.4.1 Plateforme orientée application

Cette plateforme correspond au groupe Fonctionnalité/Assocation (*Functionality/Mapping, FM*) dans [20]. Dans un premier temps, une représentation fonctionnelle du système est utilisé indépendamment d'une architecture matérielle et sans aucune notion de quantité physique comme le temps ou la puissance. Les descriptions peuvent inclure des aspects comportementaux tels que la concurrence et des concepts de communication comme les protocoles. Par la suite, la description fonctionnelle de l'application est simulée, analysée et raffinée pour être associée à une architecture matérielle. Ce raffinement comprend l'évaluation des performances et parfois la synthèse comportementale de l'architecture matérielle pour aller vers un niveau d'abstraction plus détaillé. Le manque de flexibilité de cette approche est compensé par une meilleure implémentation physique. Certains outils utilisent un à plusieurs modèles de calcul. Une plateforme orientée application s'apparente à une approche du haut vers le bas (*top-down*) car l'application guide la conception de l'architecture.

Des outils industriels comme DK Design Suite de Celoxica [14] ou BlueSpec SystemVerilog [11] appliquent cette approche mais n'offrent pas forcément un support pour des systèmes-sur-puce multiprocesseurs.

1.4.2 Plateforme orientée architecture

Cette plateforme correspond au groupe Plateforme/Association (*Platform/Mapping, PM*) dans [20]. Cette catégorie inclut les fournisseurs d'architecture matérielle avec les outils et les langages qui décrivent, manipulent et analysent ces architectures non associées à une application. Le but est d'offrir une plateforme de développement pour les développeurs d'application. Cette plateforme est donc déjà conçue et optimisée pour un domaine d'application connue, e.g. le réseau, le multimédia, la

téléphonie. Contrairement à l'approche *top-down,* l'architecture guide la conception de l'application. Il s'agit donc d'une approche du bas vers le haut (*bottom-up*). Elle se compose le plus souvent de processeurs de contrôle (processeur superscalaire), de processeurs dédiés (DSP, processeur VLIW) à une application, de blocs matériels très spécialisés (encodeur/décodeur audio, vidéo, image), de mémoires rapides (SRAM, DRAM) et de périphériques d'entrée/sortie. La plateforme offre donc une architecture MPSoC hétérogène. Chaque composant de la spécification fonctionnelle du système est associé à un élément de la plateforme sélectionnée. Cette plateforme offre une certaine flexibilité de part la diversité et la haute performance des composants présents. Il permet de réutiliser un prototype pour la même génération de produit diminuant ainsi les fais non récurrents liés à sa conception.

Le OMAP3430 [34] de Texas Instruments, le Nomadik [69] de STMicroelectronics et Nexperia [65] de Philips offrent un panel de processeurs haute performances, d'accélérateurs matériels spécialisés et de périphériques rapides d'entrée/sortie. Ces plateformes sont destinées principalement aux applications multimédia pour le téléphone mobile, la télévision numérique et le réseau.

Voici quelques outils industriels supportant l'aspect MPSoCs dans leur méthodologie et offrant l'approche architecture :

- RealView de ARM [7] qui permet avec *Core Generator* de construire des modèles SystemC abstraits. Dans [62], RealView est utilisé au niveau système pour simuler et vérifier des MPSoCs à base de processeurs ARM et de DSPs.

- CoMET et METeor de Vast [77] permet de construire plusieurs modèles virtuelles de processeurs.

- Xpres de Tensilica [74] permet de simuler plusieurs processeurs configurables Xtensa au niveau système à partir de modèle C/C++ ou SystemC.

- N2C System Designer de Coware [76] offre un environnement de cosimulation permettant la simulation de MPSoCs basé sur SytemC ou C [61].

1.4.3 Plateforme mixte

Cette plateforme correspond au groupe Fonctionnalité/Plateforme/Association (*Functionality/Platform/Mapping, FPM*) dans [20]. La conception au niveau système se divise donc en deux approches : dans le concept *top-down*, des spécifications à haut niveau de l'application aboutissent à une implémentation physique de celles-ci sur une architecture optimale alors que dans le concept *bottom-up*, une architecture matérielle existante ou un assemblage de composants existants (réutilisation de blocs IP) permet de construire l'application [64]. D'un côté, une application crée une architecture (*top-down*), de l'autre, une architecture crée une application (*bottom-up*) mais quand l'architecture est prête à recevoir une application et une application prête à être déployée sur une architecture, les deux approches se rencontrent pour former la " rencontre du milieu " (*meet-in-the-middle*). C'est à ce point qu'à lieu l'analyse de performance et l'exploration architecturale.

Ainsi, le flot de conception ESL peut se diviser en trois phases : la conception fonctionnelle, la conception architecturale dirigée par l'application et la conception architecturale orientée par plateforme. La première phase permet de créer et vérifier un modèle fonctionnelle de l'application pour tester la fonctionnalité du système. La deuxième phase permet de créer et de vérifier un modèle architectural pour trouver l'architecture optimale pour l'application répondant aux contraintes de coût et de performances. La troisième phase permet de créer un modèle plus détaillé de l'architecture pour optimiser les différents composants.

La première et deuxième phase suivent une approche *top-down* comme elles partent de l'application et la troisième phase suit une approche *bottom-up* car elle assemble

des composants IP existants.

Les plateformes Space [16] qui a servi de support au projet du présent mémoire, CoFluent Studio de CoFluent [18][64] et CoCentric System Studio de Synopsys [72] sont des exemples de plateformes mixtes supportant les trois branches du Y et l'aspect MPSoC. Une description plus détaillée de la plateforme Space est présentée dans le chapitre 3.1.

CHAPITRE 2

LES COMMUNICATIONS-SUR-PUCE

Ce chapitre présente dans un premier temps les types d'architectures de communication pour les SoCs. Par la suite, un survol des techniques d'analyse des bus et des méthodes d'exploration architecturale pour personnaliser les architectures de communication est présenté.

2.1 Les différentes architectures

Une classification des communications-sur-puce est présentée à la figure 2.1. Trois grands groupes se distinguent pour les architectures de communication. Une architecture est définie par la structure physique des interconnexions reliant les composants et par le protocole de communication qui fixe les mécanismes et les conventions d'utilisation de l'architecture par les composants [75]. En gris, apparaissent les architectures étudiées dans le présent projet.

2.1.1 Lien point à point

Dans cette architecture, des paires de composants communiquent directement par un ensemble de fils dédiés. Comme le montre la figure 2.2, cet ensemble de fils est composé des lignes de contrôle et de données. Les lignes d'adresse ne sont pas présentes car la connexion est unique. Un lien bidirectionnel nécessite donc deux liens point à point. Un composant peut avoir plusieurs liens point à point. Dans ce cas, chaque lien devient un canal de communication. Ce canal de communication

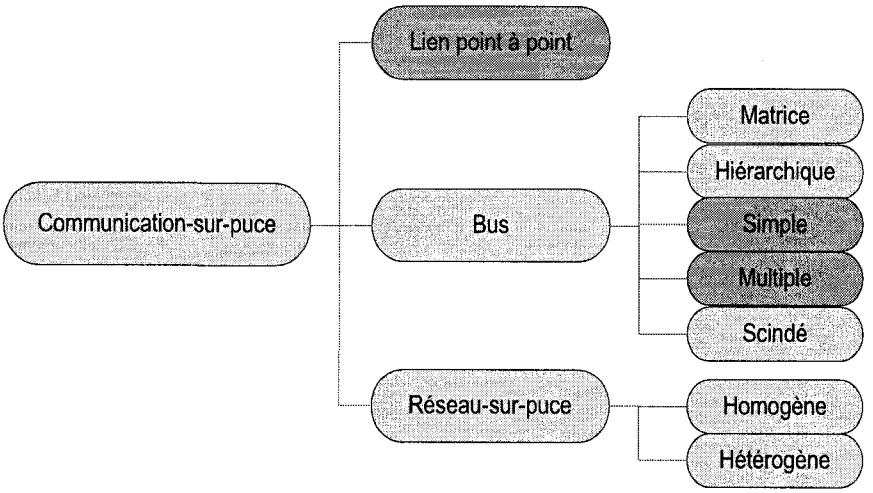

Figure 2.1 Classification des communications-sur-puce

peut stocker une ou plusieurs données et joue ainsi le rôle de FIFO. La simplicité est l'avantage des liens point à point qui peuvent être synchrone ou asynchrone. Comme le lien n'est pas partagé, la latence et la performance sont déterministes et une grande bande passante est possible entre les deux composants. Mais le câblage des liens point à point peut devenir important car plus le nombre de canal augmente, plus il faut câbler de fils [75].

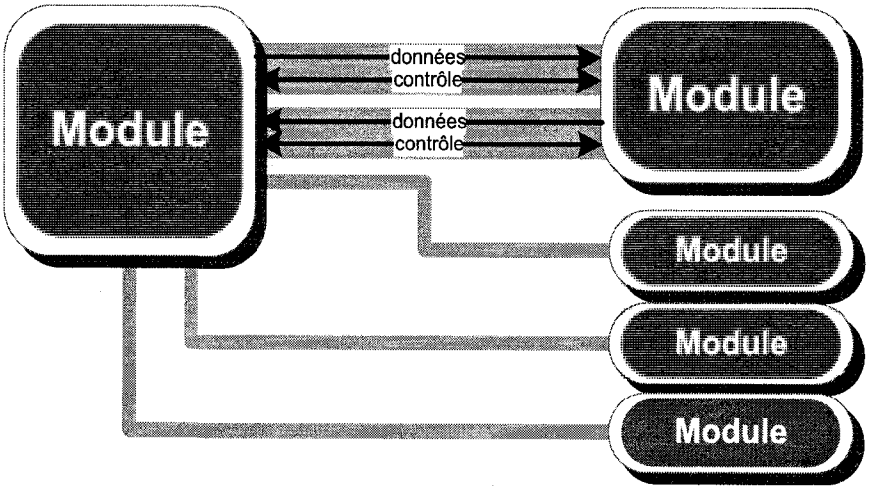

Figure 2.2 Lien point à point

Le *Fast Simplex Link (FSL)* de Xilinx [82] en est un exemple. Il permet de relier

le processeur logiciel Microblaze à des composants spécialisés. Chaque Microblaze peut avoir jusqu'à 8 liens point à point bidirectionnels soient 8 liens point à point ou canaux maîtres et 8 canaux esclaves. Le FSL permet de configurer la profondeur et le mode (synchrone et asynchrone) du FIFO.

2.1.2 Bus

Les bus sont les architectures de communication les plus répandues dans le domaine des SoCs [75][40]. Il s'agit d'une collection de fils (ou signaux) auxquels un ou plusieurs blocs IP sont connectés. Seulement, un seul bloc IP peut initier un transfert sur le bus. Dans la terminologie liée au bus, nous parlerons de :

- Maître : bloc IP initiant un transfert de données en écriture ou en lecture.
- Esclave : bloc IP qui ne fait que répondre aux transferts de données.
- Arbitre : donne l'accès au bus à un maître selon une politique d'arbitrage.
- Pont : connecte deux bus. Une des ces interfaces est maître et l'autre est esclave.

Les différentes architectures possibles sont présentées dans la figure 2.3:

1. bus simple partagé : le plus simple.
2. bus hiérarchique partagé : bus relié par un pont. Chaque bus peut avoir une structure et un protocole différent impliquant par exemple des domaines d'horloge différents et donc un stockage de données dans le pont. Souvent, un bus est dédié aux composants rapides et l'autre aux composants plus lents.
3. multibus : les composants sont reliés à plusieurs bus. Les bus ne sont pas forcément reliés par un pont.

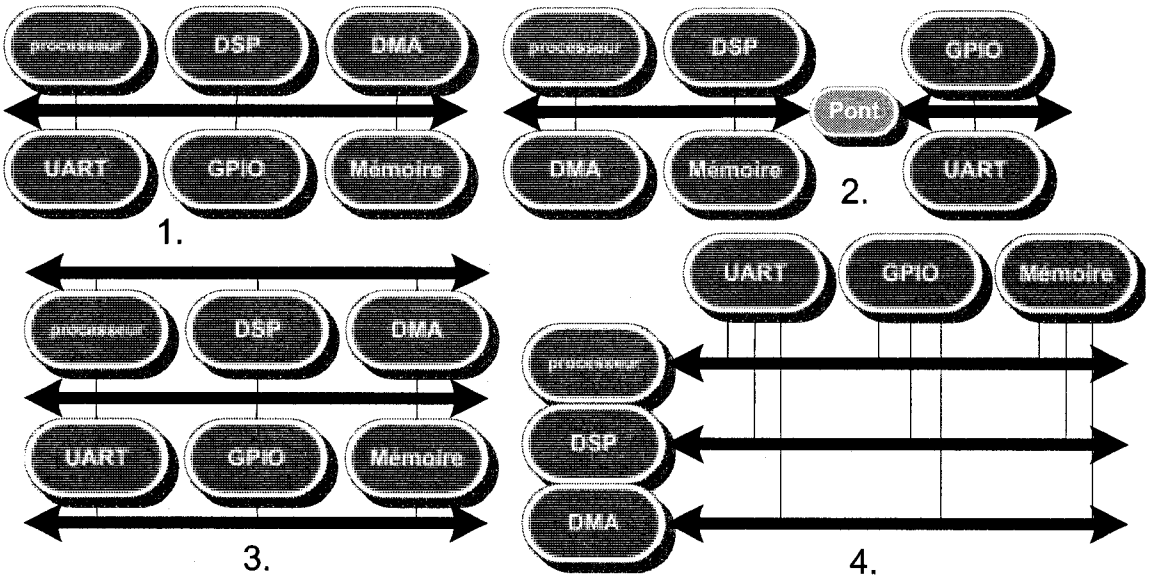

Figure 2.3 Architectures de bus

4. bus matriciel (*cross-bar*) : tous les composants maîtres sont reliés aux composants esclaves. Le principe est celui d'une matrice. Si le bus matriciel a N entrées (composants maîtres) et M sorties (composants esclaves) alors le bus a N*M liens possibles entre les maîtres et les esclaves. Ceci permet plus de parallélisme dans les communications mais nécessite de gérer l'accès aux ressources partagées. De plus, les ressources matérielles prises par cette architecture peuvent très vite augmenter.

Un bus se compose de trois ensembles ou lignes de fils appelés aussi bus : un **bus d'adresse** qui contient l'adresse du destinataire, un **bus de données** qui transportent l'information entre la source et la destination et un **bus de contrôle** pour les demandes de requête, les acquittements et des informations sur le type de transfert. Un **transfert** sur un bus implique trois phases : envoi de l'adresse et des signaux de contrôle (comprenant l'arbitrage), envoi ou réception de données, acquittement. Un bus peut être **synchrone** requérant alors une horloge pour la synchronisation des communications sur le bus, très peu de logique est requis et le bus peut fonctionner rapidement. Ou alors il peut être **asynchrone**. Dans ce cas, il

n'y a pas d'horloge, ce qui évite le phénomène de différence de temps de propagation (*clock skew*) mais un protocole de " poignée de main " (*handshake*) est obligatoire.

L'arbitrage est une pièce maîtresse dans une architecture de bus car il donne l'accès au bus à un maître pour le temps d'un transfert. L'arbitrage n'est pas requis dans un système à un maître. Un arbitrage peut être centralisé (toutes les requêtes de demande d'accès au bus arrivent au même arbitre) ou distribué (chaque maître possède son propre arbitre et chaque arbitre est relié ensemble par des fils dédiés pour décider du maître qui a le bus). Les politiques d'arbitrage ont une influence importante sur qui a le bus. Suivant la politiques, une distribution équitable ou pas de l'accès au bus s'effectue. Les plus connus sont :

- Priorité statique : un arbitre centrale accumule les requêtes de chaque maître et donne l'accès au maitre le plus prioritaire. Dans ce cas, une transaction de plusieurs mots (32 bits) ne peut être interrompue et les autres maîtres doivent attendre. À l'inverse, la transaction d'un maître de faible priorité sera interrompue.

- Tourniquet (*round-robin*) : à tour de rôle, chaque maître se voit attribuer le bus.

- TDMA (*Time Division Multiple Access*) : basé sur une roue temporelle, chaque maître se voit attribuer le bus pendant un intervalle de temps (*slot*) bien défini. Des techniques sont utilisées pour palier au problème d'intervalle de temps perdus. Notamment, en ajoutant un deuxième niveau d'arbitrage pour les intervalles perdus (e.g. priorité statique).

- Aléatoire : un maître au hasard a le bus. Il y a un risque que des maîtres ne soient jamais servis.

- Loterie : basé sur les probabilités l'algorithme donne accès à un maître qui accumule des tickets de loterie statiquement ou dynamiquement.

Les modes de transfert possibles sur un bus sont les suivants :

- Transfert simple : un maître se voit accordé le bus par l'arbitre après une demande d'accès. Les signaux de contrôle et l'adresse sont alors envoyés et les données sont envoyées en cycle subséquent.
- Transfert en rafale : le maître envoie plusieurs données en ne demandant qu'une fois le bus à l'arbitre. Le maître lève un signal spécial pour indiquer ce mode. Cela permet de sauver des cycles d'arbitrage.
- Transfert en pipeline : les phases d'adresse et de données se chevauchent. Cela ne marche que si le bus d'adresses et de données sont séparés.
- Transaction scindée ou en différé : une transaction de lecture est scindé en deux transferts. Le maître envoie une requête de lecture à l'esclave. Le maître relâche le bus et l'esclave prépare la donnée à retourner. Ici, nous gagnons en performance. Quand l'esclave est prêt, il demande le bus puis envoie les données aux maîtres.

Avantages pour la conception des SoCs :

Les bus permettent de réduire la longueur totale des fils requis dans le système et réduit aussi la surface matérielle nécessaire pour les interfaces, la communication et le contrôle. De plus, ils fournissent une ossature générique pour interconnecter des composants [75][44]. Plusieurs politiques d'arbitrage permettent d'améliorer les performances en cas de contention importante sur le bus et en fonction du type d'application. Des techniques comme les transactions scindées, le pipeline et l'utilisation de pont peuvent améliorer l'utilisation de la bande passante. De plus, l'interfaçage avec les blocs d'IPs reste simple.

Désavantages pour la conception des SoCs :

Les architectures de bus partagés sont limitées en termes d'évolutivité et de consommation de puissance. Les longs fils pour les bus ne sont pas favorables pour les technologies de l'ordre du nanomètre. De plus, avec l'augmentation de la complexité des SoCs prédit par la loi de Moore intégrant toujours plus de blocs IPs, la distribution et le partitionnement des blocs sur les bus devient souvent une tâche ad-hoc. Le bus devient un goulot d'étranglement si le nombre de composants sur le bus augmente réduisant ainsi la bande passante du bus.

2.1.2.1 Les bus standards

Il existe de nombreux standards utilisés pour établir une architecture de communication efficace. Les plus utilisés sont AMBA 2.0 et 3.0 [5] de ARM, CoreConnect [29] d'IBM, Sonics MicroNetwork de Sonics [79], STbus de STMicroelectronics [70]. D'autres bus existent comme Wishbone de Silicore Corporation [1], Avalon d'Altera ou CoreFrame de PalmChip [60]. Dans cette section, le protocole CoreConnect (utilisé dans le présent projet) et AMBA sont décrits un peu plus en détails. Pour une comparaison des différents bus selon certaines caractéristiques expliqués précédemment, le lecteur est invité à consulter l'annexe II.

Des études sur différents types de bus [44], bus hiérarchique, bus matriciel, bus global et bus avec mémoire tampon, testés avec des applications réelles montrent que les bus sont viables pour le domaine des MPSoCs.

AMBA 2.0 et 3.0 :

Ces deux versions proposent trois types de bus pouvant être utilisés en bus hiérarchique comme le montre la figure 2.4 (tiré de www.synopsys.com):

Le protocole AMBA 2.0 spécifie deux bus : le bus AHB (*Advanced High-performance Bus*) destiné aux composants rapides comme des processeurs, des DMA et le bus APB

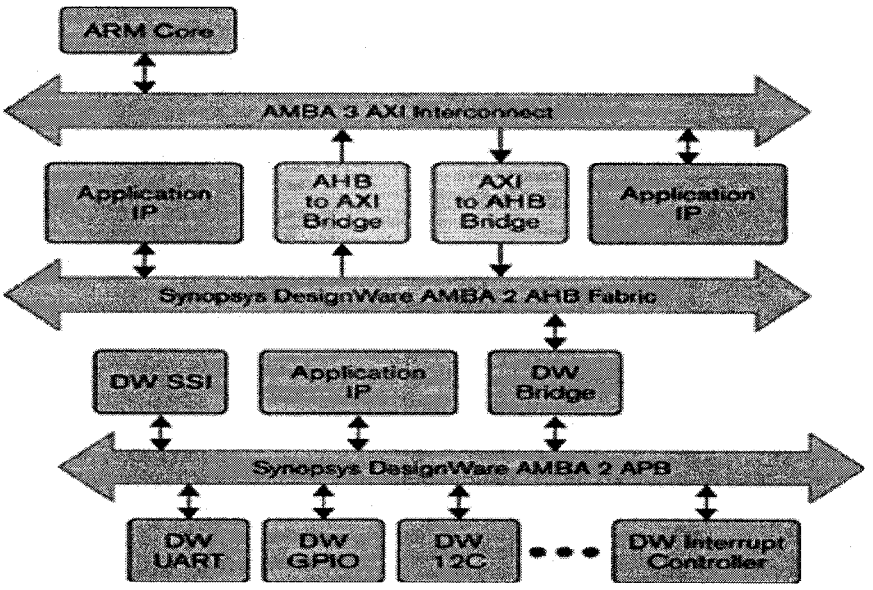

Figure 2.4 Architecture typique avec les bus AMBA

(*Advanced Peripheral Bus*) destiné à des périphériques esclaves plus lents comme un UART, une minuterie. Le protocole AMBA 3.0 introduit un bus de haute performance, AXI, utile dans des systèmes travaillant à haute fréquence.

CoreConnect d'IBM :

Ce protocole propose un découpage hiérarchique en trois bus : un bus rapide pour des processeurs, des périphériques rapides, des DMAs, le PLB (*Processor Local Bus*), un bus pour des périphériques plus lents, le OPB (*On-chip Peripheral Bus*) et un bus reliant les composants pour du contrôle, du diagnostique et de la gestion d'erreur, le DCR (*Data Control Register*). Une structure typique est montrée dans la figure 2.5.

Le bus PLB répond à des problèmes de faible latence, de haute performance et de flexibilité rencontrés dans la conception de SoC en offrant :

- Une configuration du bus en 32, 64 ou 128 bits;
- Un bus de données d'écriture et de lecture séparé pour des transferts chevauchés augmentant la bande passante, et un bus d'adresse avec possibilité de transac-

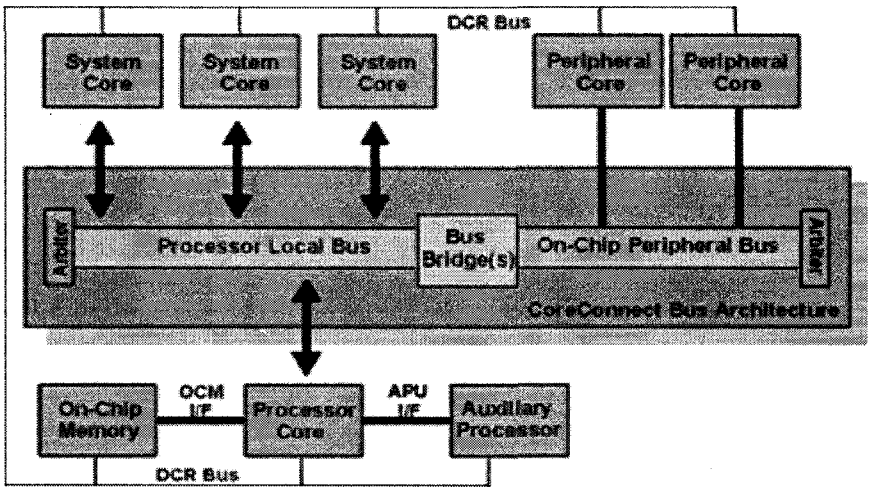

Figure 2.5 Architecture typique avec les bus CoreConnect

tions scindées;

- Taux de transfert de données élevé grâce à :
  - Transferts en mode rafale de taille variable ou fixe
  - Pipeline dans l'arbitre permettant des transferts en écriture et en lecture concurrente
  - Transactions scindées
  - Transferts DMA
  - Transferts de ligne de cache
  - Chevauchement de cycle d'arbitrage
  - Arbitrage de plus haute priorité ou LRU (Least Recently Used)

Le bus OPB sert à réduire la charge capacitive du bus PLB en accueillant des périphériques avec un faible débit tels que des ports série, parallèles, des UARTs, des minuteries, des entrées/sortie d'usage général. Il offre les caractéristiques suivantes :

- Protocole synchrone avec des bus d'adresse et de données (32 bits ou 64 bits chacun) séparées;

- Dimensionnement dynamique du bus pour supporter les transferts de 8, 16 et 32 bits;
- Mode séquentiel (équivalent à un mode rafale);
- Plusieurs maîtres OPB possibles;
- Support optionnel pour l'octet valide (Byte Enable);
- Timeout de 16 cycles fournit par l'arbitre permettant à une requête de ne pas geler le bus et pouvant être désactivé par l'esclave;
- Les maîtres peuvent bloquer le bus par le signal bus lock;
- L'esclave peut demander un retry pour signaler à un maître qu'il n'est pas prêt;
- Cycle d'arbitrage chevauché avec le dernier cycle de transfert pour améliorer bande passante;
- Mode parcage pour optimiser l'utilisation du bus quand le bus n'est demandé par aucun maître.

Le bus DCR est utilisé pour lire et écrire les registres de statut et de configuration. Il fournit un débit maximum de un transfert d'écriture ou de lecture tous les deux cycles. Il est synchrone et implémenté par des multiplexeurs distribués. Il utilise une structure de bus en anneau pour relier tous les composants et minimiser ainsi la surface de silicium utilisée.

2.1.3 Réseau-sur-puce

Les réseaux-sur-puce (*network-On-Chip, NoC*) s'inspirent des grands réseaux de communication comme les réseaux locaux (LAN) ou les réseaux plus large (*WAN*) où les communications inter-processeurs sont supportées par des paquets routés à

travers des commutateurs (*switch).* Les communications s'effectuent en envoyant des paquets de message entre les blocs IPs à travers le réseau de commutateurs. Beaucoup de recherche est faite pour adapter les concepts des réseaux sur Internet au monde des SoCs [8].

Les réseaux-sur-puce sont caractérisés par leur topologie et leur protocole de routage. La topologie définit la géométrie des liens de communication alors que le protocole gouverne l'usage de ces derniers. Beaucoup de combinaisons de topologie/protocole existent pour obtenir la meilleure communication. La performance d'un réseau est mesuré de manière quantitative par la latence, le débit, la consommation de puissance et l'utilisation de surface et de manière qualitative par la reconfigurabilité (dynamique ou statique), l'évolutivité, la qualité de services. Elle est influencée par la topologie, les techniques de routage, le flot de contrôle, l'architecture des routeurs (commutateur ou mémoire tampon)

La terminologie du réseau-sur-puce définit les termes suivants :

- Canal : transporte physiquement l'information.
- Commutateur : permet de router l'information grâce à des protocoles de routage en établissant des liens entre les canaux.
- Nœud terminal : représente un composant (processeur, mémoire, système basé sur un bus avec plusieurs composants) connecté au réseau via un commutateur.

Un réseau s'inspire du modèle OSI (*Open System Interconnexion*) à sept couches. Les couches pour un réseau-sur-puce sont au nombre de quatre correspondant aux quatre plus basses couches du modèle OSI [8].

Une topologie réfère donc à l'arrangement statique des canaux et des nœuds. Il existe deux types de topologie [75]:

- homogène : les canaux et les nœuds sont disposés de façon régulière (topologie en anneau, topologie torus, topologie en grille)
- hétérogène : les interconnexions sont de types point à point

Avantages pour la conception des SoCs :

Les NoCs permettent d'éliminer les long fils à partir du moment que les commutateurs sont connectés de façon régulière sur la base de lien point à point entre les canaux. Le concept de couches permet de séparer les transactions et le support physique, ce qui conduit à optimiser ces deux éléments de façon indépendante [75]. Un NoC est évolutif par rapport aux nombre de nœuds terminals. Les composants du réseau peuvent être développés indépendamment avant d'être connectés ensemble. Des débits de l'ordre la dizaine de Go/s peuvent être atteints avec des fréquences de plus de 750 MHz [8].

Désavantages pour la conception des SoCs :

Des développements de NoCs sur technologie FPGA [9][2] montrent que la conception des commutateurs et l'implémentation des algorithmes de routage provoquent une consommation excessive de surface du FPGA. Les latences de communication entre les blocs IP peuvent être importantes [8]. De plus, les interfaces entre les blocs IPs et le réseau sont plus compliquées à concevoir que pour les bus. Enfin, de nouvelles méthodologies pour développer, tester et intégrer des NoCs sont requises.

À l'heure actuelle, les communications-sur-puce à base de bus dominent le marché face aux NoCs. Ces derniers représentent en revanche une voie intéressante et prometteuse pour le futur des MPSoCs.

2.2 Techniques d'analyse

De nombreuses recherches ont été menées afin de proposer des techniques d'analyse pour estimer l'impact du réseau de communication sur la performance globale du système et la consommation d'énergie. Ainsi, trois groupes de techniques se distinguent pour permettre d'aiguiller la sélection, la conception et l'optimisation de ces réseaux [39].

- Techniques basées sur la **simulation** : les effets des réseaux de communication sont étudiés lors de la simulation complète du système comprenant des modèles de réseaux de communication de différente topologie et protocole. Ces techniques sont souvent inadaptées à l'exploration de grands systèmes comme ceux offerts par les réseaux de communication existants (bus standards) et émergeant (réseaux-sur-puce). Pour obtenir des vitesses de simulation rapides, des modèles abstraits des composants et de réseaux de communication sont utilisés. Mais l'abstraction des détails de communication peut nuire à la précision des résultats au profit de l'efficacité de la simulation. Dans [50], les auteurs proposent un environnement de simulation qui modélise le matériel et le logiciel avec SystemC à un haut niveau d'abstraction. Leur environnement offre de bonne vitesse de simulation au niveau 'précis au cycle' (*cycle accurate*) et 'précis au signal' (*signal accurate*) pour des applications MPSoCs significatives. Ils génèrent un trafic fonctionnel pour un réseau de communication comme AMBA ou STBus qui relient des processeurs (e.g. processeur ARM). Les statistiques et les estimations de la performance sont réalistes car ils se basent sur un trafic fonctionnel et non sur des traces d'exécution fixe, des modèles analytiques ou des générateurs statiques. [25] est un autre exemple dans lequel des modèles de réseau de communication en SystemC sont intégrés à un environnement de cosimulation. Leur environnement offre trois niveaux d'abstraction à travers lesquels le réseau est raffiné vers une architecture

existante (e.g. bus) permettant différents degrés de précision.

- Techniques basées sur **l'estimation** : des modèles statiques des réseaux de communications sont utilisés, soit pour les latences [43], soit pour la consommation de puissance [41]. Ces techniques font l'hypothèse que les communications et les calculs peuvent être statiquement ordonnancés. Ceci peut engendrer de l'imprécision dans les résultats de simulation car les effets dynamiques comme l'attente du à la contention sur le bus ne sont pas pris en compte. Dans [43], des équations permettent d'estimer l'influence des divers paramètres d'un réseau de communication (e.g. taille du bus, transferts en rafale ou pas, fréquence, etc) à travers un partitionnement matériel/logiciel.

- Techniques basées sur les **traces** : les effets des réseaux de communication sont intégrés dans une analyse au niveau système en étudiant les traces d'une exécution. Une cosimulation initiale du système à partir de modèles abstraits de composants est réalisée. Par la suite, les traces de cette simulation sont envoyées vers un outil d'analyse qui en fonction d'un réseau de communication spécifié estiment la performance du système. [46] utilise une méthodologie avec une simulation initiale du système avec des communications abstraites (e.g. évènements ou transferts de données abstraits). Une extraction des traces de simulation contenant des informations suffisantes sur les opérations de calcul et de communication est analysée sous forme de graphe. En spécifiant la topologie du réseau de communication, son positionnement dans les divers chemins du système et en choisissant une certaine configuration du protocole, le graphe estime la performance du système et fournit des statistiques sur les composants et leur communication.

2.3 Méthodes d'exploration architecturale

L'exploration architecturale des réseaux de communication couvre différents aspects.

- Choisir la topologie du réseau. Combien de bus ? Quelle géométrie d'interconnexion (bus simple, hiérarchique, matriciel, spécifique à une application) ? Comment répartir les blocs IP sur les différents bus ?
- Configurer le protocole du bus. Par exemple, politique d'arbitrage pour les bus partagés, la taille des bus, la fréquence des bus synchrones, la taille des transferts en rafale, etc.

L'objectif de cette exploration peut varier en fonction des spécifications de l'application. Souvent, la topologie et la configuration idéale du réseau fait intervenir des compromis entre un ou plusieurs de ces critères : performance, puissance, coût (nombre de bus et logique associé), surface, implémentation physique, etc.

Ainsi, les diverses méthodes d'exploration des réseaux de communication existantes s'emploient à mettre l'accent sur la topologie, la configuration du protocole, les deux. Certaines techniques tiennent compte de l'aspect physique réel des bus [22].

Par exemple, les auteurs de [45] mettent l'accent sur la topologie. Grâce à leur outil, *BusSynth*, il synthétise cinq architectures de bus différents et permettent l'exploration de ces différentes architectures en se basant sur des facteurs de performance comme le type de processeur, le style de programmation logicielle.

[47] propose une technique pour optimiser le protocole du réseau de communication afin de mieux correspondre au trafic généré par une application. Ainsi, en proposant des patrons de réseaux de communication (bus hiérarchique, bus matriciel, etc), leur méthode permet d'explorer diverses configurations des paramètres du protocole (taille des bus, arbitrage, taille maximale du mode rafale) pour n'importe quel

type d'application. Par plusieurs itérations, l'exploration mènera à la meilleure configuration du protocole.

Dans [63], l'exploration architecturale permet de construire des topologies à base de bus matriciel avec diverses configurations du protocole (choix de la politique d'arbitrage, taille du bus, fréquence de fonctionnement). Ainsi, cela permet de réponde aux contraintes de débit et de minimiser le nombre de bus dans la matrice.

De plus, certaines techniques focalisent plus particulièrement sur l'exploration à partir de l'estimation et de l'analyse de la consommation de puissance du réseau de communication. C'est le cas de [15] qui analyse une architecture de bus impliquant le scindement de ce bus en plusieurs segments avec des circuits pilotes double port. Leur méthode permet de réduire la charge capacitive des bus, économisant ainsi de l'énergie. En formulant le problème du scindement du bus en segments selon un problème d'échange de données interbus minimum et en utilisant des algorithmes heuristiques, il démontre le gain de réduction de puissance dissipée sur des bus standards comme AMBA de ARM et CoreConnect d'IBM.

Ainsi s'achève la revue de littérature qui met en lumière les défis rencontrés dans la conception de systèmes-sur-puce multiprocesseur tant au niveau logiciel que matériel. Sur le plan matériel, nous voyons que l'exploration des réseaux de communication est très importante car sans médium de transport, il n'y a pas d'échanges inter-processeurs possibles mais avant tout, un réseau de communication optimisé pour le trafic généré par l'application permet de répondre efficacement aux contraintes initiales. La partie suivante décrit les différents modèles abstraits créés en SystemC pour explorer différentes topologies de bus en focalisant sur les effets de différents paramètres du protocole. La technique utilisée est basée sur la simulation du système entier au niveau TF et BCA de la plateforme Space.

CHAPITRE 3

EXPLORATION ARCHITECTURALE DES COMMUNICATIONS SUR PUCE

Ce chapitre donne des précisions sur la plateforme ESL Space, notamment sur les différents niveaux d'abstraction (UTF, TF et BCA). Par la suite, la méthode d'exploration multi-niveaux d'abstraction des architectures de communication est décrite et enfin les composants de haut niveau développés en SystemC qui permettent d'explorer ces architectures de communication au niveau TF et BCA de Space sont présentés. Le lecteur est invité à consulter la partie sur SystemC de l'annexe I pour avoir en tête la terminologie associée à SystemC et utilisé dans la description des composants abstraits.

Note : dans cette section, un vocabulaire lié aux communications sur les réseaux de communication sera utilisé. Afin d'éviter les confusions, la terminologie suivante est utilisée.

- Transaction ou message : représente un échange entier de données entre deux entités. Se compose de un ou plusieurs transferts. Par exemple, une transaction Space se compose d'un transfert d'entête et de un ou plusieurs transferts de données.

- Transfert ou requête : envoi de données selon plusieurs phases (arbitrage, transfert, acquittement). Nous parlerons de requête d'écriture ou de lecture.

- Phase : représente une étape particulière dans le traitement du transfert. Il manipule un groupe de signaux (e.g. signaux d'arbitrage, signaux de contrôle, signaux d'adresse, de données)

- Maître : désigne un OPB adaptateur, un processeur, un DMA.

- Esclave : désigne un périphérique (e.g. mémoire, UART) ou un OPB adaptateur.

- Module : au sens de SystemC, un module désigne une entité possédant un processus. Par conséquent, un module peut désigner un maître ou un esclave.

3.1 La plateforme Space

La plateforme Space [16][57] propose une exploration architecturale à l'aide d'une méthodologie de raffinement matériel/logiciel. Elle répond aux objectifs suivants :

1. Permettre la simulation et l'analyse de performances d'un système grâce à des modèles abstraits au niveau système;

2. Permettre une exploration architecturale rapide en testant plusieurs partitionnements logiciel/matériel afin de trouver l'architecture matérielle optimale pour l'application;

3. Produire une architecture matérielle de haut niveau incluant l'application logicielle servant de référence pour la conception RTL.

La méthodologie de Space repose sur trois niveaux d'abstraction qui offre un raffinement progressif du système en termes de détails. Le concepteur peut revenir à un niveau précédent au besoin. L'appellation des niveaux suit le modèle OSCI $1^{ère}$ génération (Figure 1.2).

- 1^{er} niveau (UTF) : le concepteur spécifie l'application et valide la fonctionnalité avec le simulateur de SystemC. Les communications sont considérées comme des messages sans notion de temps. Le caractère bloquant et non-bloquant des messages assurent la synchronisation ente les modules.

- $2^{ème}$ niveau (TF) : le concepteur partitionne l'application en modules logiciels et matériels. La simulation de la partie matérielle s'effectue avec le simulateur

de SystemC et la partie logicielle est exécutée par le RTOS encapsulé dans l'interface SystemC/RTOS. Des annotations temporelles sont ajoutées dans les modules et le canal pour refléter des latences de calcul et de communication.

- $3^{ème}$ niveau (BCA) : les modules précédemment testés au niveau UTF et TF sont réutilisés tel quel et placés dans une architecture matérielle précis au cycle (*cycle accurate*). L'utilisateur teste ses partitionnements sur une architecture se rapprochant d'une architecture réelle. Notamment, les modèles abstraits de Space s'appuie sur le protocole CoreConnect d'IBM pour les bus et sur les blocs IP développés par Xilinx et compatibles avec le standard CoreConnect. Le RTOS du niveau TF s'exécute maintenant sur un simulateur de jeu d'instruction (*Instruction Set Simulator, ISS*) représentant le comportement d'un processeur (e.g. MicroBlaze, PowerPC). Cet ISS est considéré comme un module SystemC et est donc pris en charge par le simulateur SystemC.

La figure 3.1 montre le raffinement offert par les différents niveaux d'abstraction. Au niveau TF et BCA, il est possible de surveiller les bus, les mémoires, les processeurs ainsi que le système d'exploitation qui s'exécute dessus pour aider à déboguer et partitionner un système [57].

3.1.1 Types de communication

Les communications offertes par la librairie de Space reposent sur quatre fonctions : *ModuleRead(), ModuleWrite(), DeviceRead()* et *DeviceWrite()*. Les deux premiers servent à une communication entre module et les deux autres à une communication entre un module et un périphérique. La figure 3.2 montre le fonctionnement des quatre fonctions.

Le cas **1)** concerne une écriture associée à une lecture d'un module à un module. Lors d'une écriture, la transaction est envoyée au destinataire en plusieurs transferts.

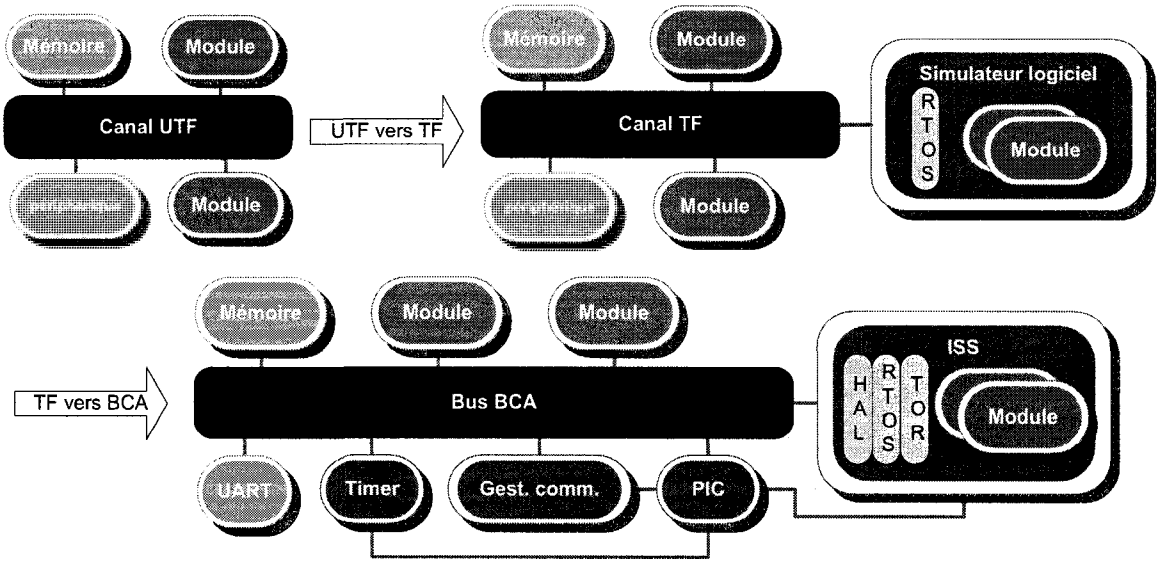

Figure 3.1 Raffinement à travers les niveaux de Space

Les transferts sont accumulés dans le module adaptateur. Chaque transaction s'accompagne d'un entête Space permettant au module destinataire de reconstituer la transaction reçue par morceaux. Notez que les lectures effectuées par un module sont locales. Elles ne passent pas sur le canal de communication. Lors de la lecture, une analyse de la transaction reçue permet l'envoi ou non d'un acquittement Space pour débloquer le module utilisateur expéditeur. L'acquittement Space constitue un message à part entière et est lié au caractère bloquant des messages de Space. Le caractère bloquant ou non-bloquant assure le synchronisme entre les modules.

Le cas **2)** concerne une lecture et une écriture d'un module dans un périphérique. Il s'agit d'une séquence d'appel de fonctions à travers les ports et les interfaces de différents composants. Le périphérique reproduit l'action d'écriture ou de lecture d'une ou plusieurs données (e.g. écriture ou lecture d'une case mémoire, d'un registre dans une minuterie, un gestionnaire d'interruption ou un UART). Les messages pour un périphérique ne nécessitent pas d'entête et un périphérique n'initie pas de transaction.

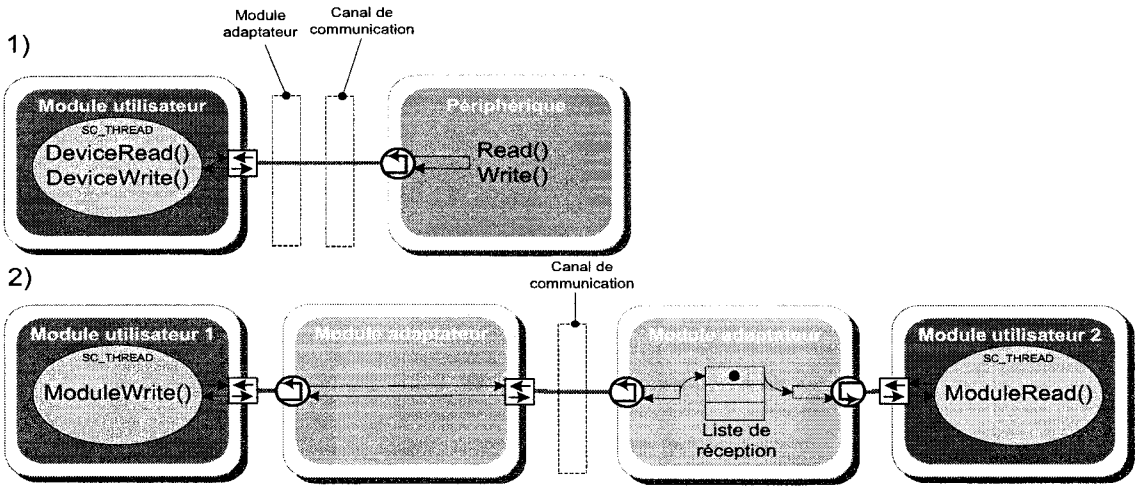

Figure 3.2 Types de communication dans Space

3.2 Méthodologie d'exploration

Afin d'exploiter les composants abstraits décrits précédemment, une méthode d'exploration d'architecture de communication à travers les trois niveaux de SPACE est proposée (figure 3.3).

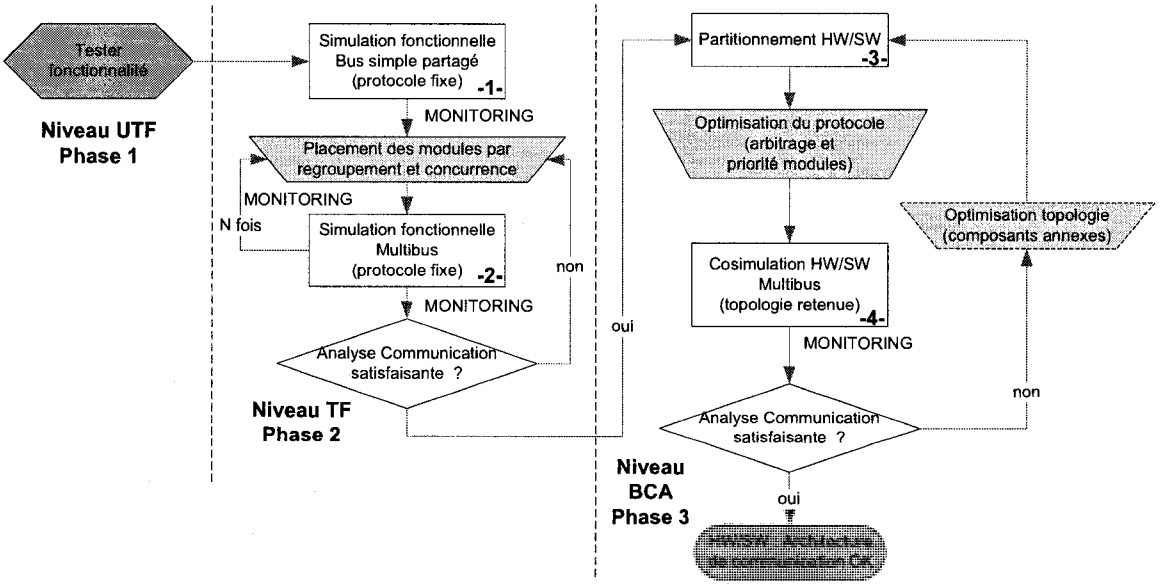

Figure 3.3 Méthode d'exploration des architectures de communication dans Space

Niveau UTF : les fonctions de l'application sont testées sans supposer une quelconque architecture de communication.

Niveau TF : l'exploration commence par une simulation fonctionnelle avec une architecture initiale basée sur un bus simple partagé (étape -1-). Le protocole du bus (arbitrage, taille du bus) est fixé à des valeurs par défaut. La performance de cette architecture sera la référence à améliorer. Il est rare que l'architecture bus simple partagé suffise dès le début à une application complexe dû à la forte contention sur le bus. Grâce au *monitoring* offert par SPACE, les modules vont être placés sur un réseau multibus selon la technique de **regroupement** et de **concurrence**. En effet, pour répartir les modules au mieux, il faut prendre en compte deux critères : les composants qui communiquent beaucoup ensemble sur le même bus (regroupement = *clustering*) et les composants qui peuvent s'exécuter parallèlement sur deux bus différents (concurrence). Cela va permettre de limiter les communications **inter-bus** au profit des communications **intra-bus**. Plusieurs simulations matérielles avec différents réseaux multibus sont lancées (étape -2-). Après chaque simulation multibus, un nouveau placement des modules sur les bus est effectué. Cette opération peut être répétée N fois avec N choisi par l'utilisateur (e.g. 3). Ensuite, les performances apportées par les différentes configurations multibus sont comparées à l'architecture de référence. Si l'analyse des communications montre une architecture multibus qui améliore la performance, nous pouvons passer à la phase 3. Sinon, nous pouvons recommencer au placement des modules et relancer l'étape -2-.

Niveau BCA : nous arrivons ici avec une architecture multibus retenue qui devient la nouvelle référence à améliorer. Les modules de l'application vont être séparés en modules logiciels ou matériels, c'est l'étape de partitionnement matériel/logiciel (HW/SW, étape -3-). Le partitionnement HW/SW peut aboutir à une solution toute matérielle, toute logicielle ou à un mélange des deux. Un partitionnement donné peut influencer directement l'architecture de communication. Par exemple, dans

une configuration multiprocesseur à base de MicroBlaze, il ne peut y avoir qu'un seul processeur par bus OPB. Le partitionnement est une étape importante dans la conception des SoCs et de très nombreuses recherches sont développées pour aider au partitionnement. Cependant, le partitionnement n'étant pas le sujet de ce mémoire, nous considèrerons un partitionnement déjà établi de façon *ad hoc* ou à l'aide de technique d'aide au partitionnement [56] pour l'application testée.

L'architecture de communication va être optimisée par la configuration du protocole (largeur du bus, arbitrage) des différents bus[1]. Un jeu de paramètres du protocole est choisi. Une cosimulation matérielle/logicielle (HW/SW) de l'architecture multibus optimisée est lancée (étape -4-). Grâce au *monitoring*, la performance du système est analysée. Si celle-ci n'est pas satisfaisante, nous pouvons essayer un nouveau jeu de paramètres du protocole. Bien évidemment, un nouveau partitionnement HW/SW (étape -3-) pourrait être choisi grâce à la possibilité de SPACE de déplacer un module matériel en logiciel et inversement sans recoder le module. Mais avec le risque de modifier l'architecture de communication surtout pour un système multiprocesseur. Néanmoins l'architecture multibus améliorée peut rester valide car SPACE s'occupe d'assurer l'intégrité des communications. E.g. un lien point à point entre deux modules matériels deviendra un lien point à point entre un processeur et un module matériel si l'autre a été mis en logiciel.

Il peut arriver qu'à ce niveau, l'utilisation des ponts OPB-OPB révèle des risques de famine ou d'interblocage (cf. partie 4.4). Il est possible d'utiliser les composants annexes, i.e. lien point à point et/ou mémoire double port, pour résoudre les problèmes de l'architecture multibus (trapèze en pointillé sur la figure 3.3). Dans le cas où la performance est satisfaisante, l'architecture de communication optimisée est alors définitivement retenue pour le partitionnement choisi.

Note : dans la phase 2, N peut devenir très grand pour des systèmes avec beaucoup de

[1] Au moment de l'élaboration de la méthode, seule une architecture homogène de bus OPB était disponible au niveau BCA.

modules indépendants. Pour cette raison, l'exploration de N configurations multibus à comparer à l'architecture de bus simple est adaptée à des applications moyennement complexes où N peut valoir entre 3 et 5 configurations multibus différentes.

3.3 Les composants au niveau TF

Ce niveau transactionnel offre la possibilité au concepteur d'avoir une première estimation des latences de communication et de calculs. La simulation rapide permet d'explorer plusieurs réseaux de communication et d'analyser des résultats post-simulation grâce au " monitoring " (e.g. utilisation du bus, nombres d'accès mémoires pour chaque module, communications entre chaque module). La figure III.1 de l'annexe III décrit l'ensemble des composants du niveau TF.

3.3.1 Le canal TF

3.3.1.1 Caractéristiques

Ce canal permet de relier des modules utilisateurs et des périphériques ensemble. Un seul module peut avoir accès au canal pour communiquer avec un autre module ou un périphérique. Le canal est appelé **fonctionnelle avec notion de temps** (*Timed Functional, TF*) car chaque transaction sur le canal fait intervenir différentes latences et les détails d'un protocole de bus ne sont pas connus. Seul des fonctionnalités communes à plusieurs bus standards sont modélisées. Ainsi, une transaction est divisée en trois phases : une phase d'arbitrage, une phase de décodage d'adresse et une phase de transfert de données.

Le canal TF inclut un arbitre qui donne l'accès au bus à un seul module en fonction de différentes politiques d'arbitrage. Il est possible de configurer le canal TF sans

arbitrage dans le cas où un seul maître est relié au canal.

Le canal TF s'inspire du modèle transactionnel précis au cycle, *simple bus* [59], fournit avec les sources de SystemC 2.1. Le *simple bus* ne modélise pas de pipeline, de transactions scindées, ni de schéma requête/acquittement. Il possède aussi un arbitre avec lequel il communique via un port. Le canal TF ne modélise pas non plus de technique de pipeline de données qui améliore la bande passante d'un bus. Le canal TF est configurable de façon à reproduire un maximum de bus standard. À partir des caractéristiques générales d'un bus (partie 2.1.2), le canal TF offre les paramètres de configuration suivants :

- Fréquence de fonctionnement
- Taille du bus (8, 16, 32, 64, 128 bits)
- Politique d'arbitrage (FIFO, plus haute priorité, sans arbitrage)
- Latence d'arbitrage (nombre de cycles d'horloge pour arbitrer une requête)
- Latence de décodage d'adresse/transfert de données
- Latence de transfert de données/acquittement
- Accès supplémentaire (e.g. 1 accès de plus pour l'entête Space)
- *Mode rafale ou non*

Le canal TF est donc capable de reproduire les latences d'un transfert de base de cinq bus industriels : AHB (*Advanced High Bus*), ASB (*Advanced System Bus*), APB (*Advanced Peripheral Bus*) du protocole AMBA (ARM) et PLB (*Processor Local Bus*), OPB (*On-chip Peripheral Bus*) du protocole CoreConnect (IBM). En examinant la documentation technique des ces différents bus [4][31][30], quelque soit le type de bus, un transfert se découpe selon les phases de la figure 3.4.

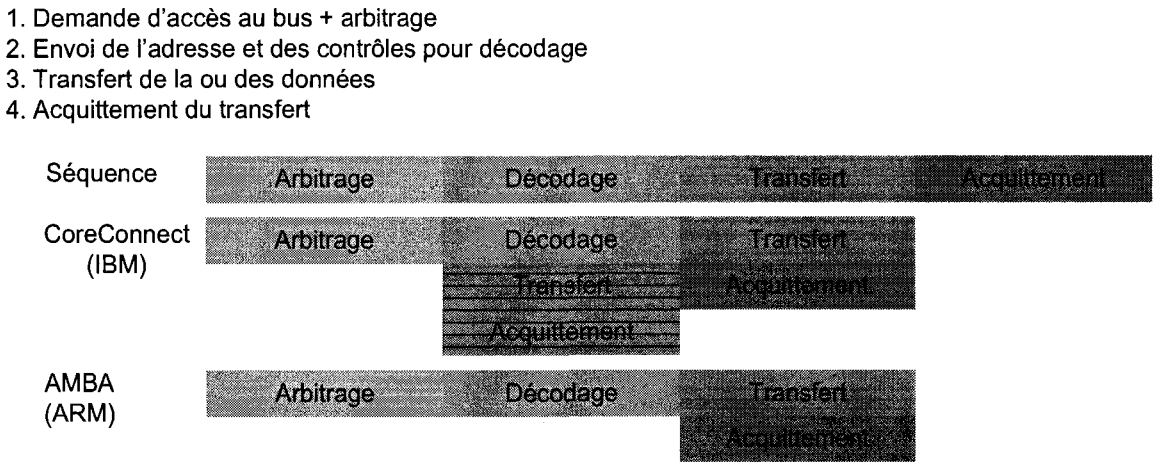

Figure 3.4 Phases d'un transfert de données sur un bus

Un ou des maîtres demandent l'accès au bus à l'arbitre. Celui-ci choisit le maître selon une politique d'arbitrage (1.). L'arbitrage peut prendre de 1 à 3 cycles selon le protocole AMBA ou CoreConnect. Ensuite, une fois le bus accordé, le maître envoie l'adresse de l'esclave et les signaux de contrôles servant à sélectionner le bon esclave (2.). Le décodage prend 1 cycle quelque soit le protocole. Au cycle suivant, le maître envoie la donnée à écrire ou reçoit une donnée de l'esclave (3.). Le transfert prendre un 1 cycle pour un transfert simple. L'esclave acquitte toujours le transfert pour donner un statut sur la réussite de l'opération (4.). L'acquittement dure toujours 1 cycle quelque soit le protocole. La figure 3.4 montre que l'étape de transfert et d'acquittement se passent en même temps. En effet, un cycle peut suffire pour acheminer la donnée, que l'esclave la traite et qu'il retourne un acquittement. Évidemment, dans le cas d'esclave plus lent, l'acquittement peut prendre plusieurs cycles avant d'avoir lieu. Par exemple, dans le protocole AMBA, l'esclave peut ajouter des états d'attente (*wait state*) si celui-ci a une latence élevée. Dans le cas du bus OPB, la spécification indique que des esclaves avec une latence d'un cycle et un décodage d'adresse combinatoire, il est possible d'atteindre 1 cycle pour le décodage/transfert/acquittement (partie rayée sur la figure 3.4). Un tableau en annexe IV indique les différentes latences en fonction des bus.

3.3.1.2 Fonctionnement interne

La figure 3.5 montre le diagramme de classe du canal TF en gris foncé ainsi que sa représentation SystemC.

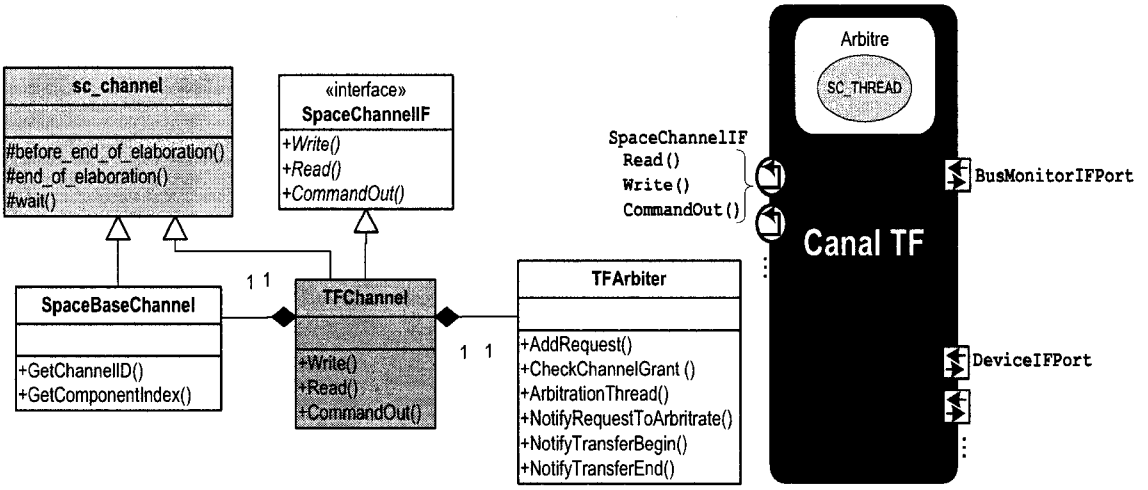

Figure 3.5 Diagramme de classe du canal TF

Le canal TF appelé *TFChannel* dérive de *SpaceChannelIF* et de la classe de base SystemC, *sc_channel*. Cette dernière est simplement une redéfinition de la classe *sc_module* qui fournit les méthodes de base pour la phase d'élaboration et de simulation de SystemC. L'interface *SpaceChannelIF* fournit les méthodes de communication *Read()*, *Write()* et *CommandOut()* dont la fonctionnalité est implémentée dans le canal TF. Le canal TF est composé d'un arbitre, classe *TFArbiter* qui contient un processus (*thread*) de type SC_THREAD qui s'occupe de réaliser l'arbitrage du ou des requêtes d'accès au canal. Le canal est aussi composé d'une classe de base de la librairie Space, *SpaceBaseChannel*, qui fournit des méthodes pour connaître le numéro d'identification du canal (ID servant pour des fins de *monitoring*) et retrouver le port destinataire dans le tableau de ports. Le canal TF possède autant de port, *DeviceIFPort*, que de composants connectés. Ces ports sont regroupés dans un tableau. À travers un port *DeviceIFPort*, le canal TF appelle les fonctions *Read()*, *Write()* et *CommandFromChannel()* implémentées par le composant (périphérique

ou module adaptateur) et reproduisant une certaine fonctionnalité associée aux composants (un accès mémoire en écriture ou lecture, une écriture pour un module, un acquittement pour une écriture bloquante d'un module). Le canal TF possède aussi un port *BusMonitorIFPort* qui sert au *monitoring* afin de recueillir des statistiques sur les communications. Ces statistiques serviront par exemple à connaître l'utilisation du bus, sa bande passante, les types de communication entre des modules, le temps passé dans les communications, etc.

Déroulement d'une communication :

L'exemple 3.6 d'une lecture d'un module dans une mémoire illustre le principe de déroulement d'une communication à travers le canal TF.

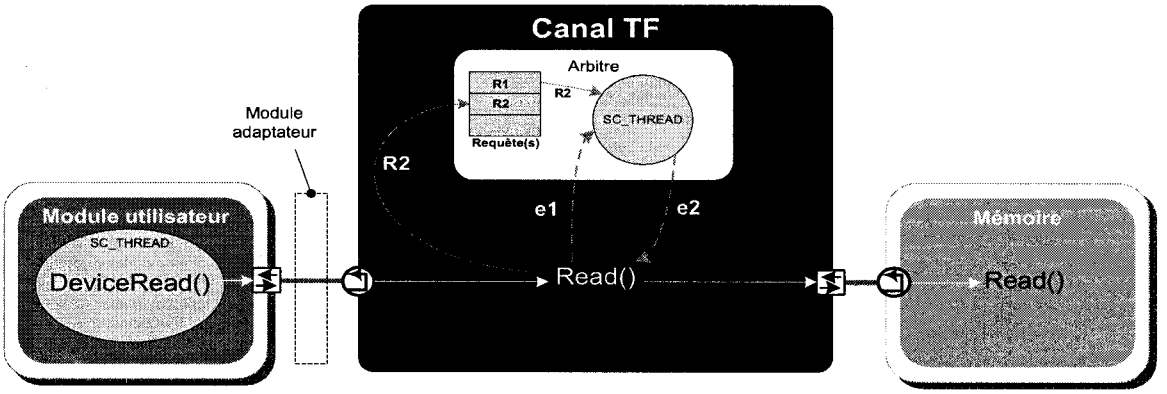

Figure 3.6 Communication sur le canal TF

Considérons un module utilisateur qui lit une mémoire. Il utilise la fonction de communication *DeviceRead()*. Cette méthode à travers le port du module appelle la fonction *Read()* du module adaptateur qui à son tour appelle la fonction *Read()* implémentée par le canal TF. Dans la fonction *Read()* du canal, les étapes suivantes s'effectuent :

1. la requête **R2** de ce module est stockée dans une liste de requêtes de demande d'accès contenue dans l'arbitre. Si plusieurs modules initient une transaction sur le canal, toutes les transactions sont stockées dans cette liste. Sur le schéma,

il y a une autre requête **R1** d'un autre module en attente d'arbitrage.

2. l'arbitre contient un processus qui attend sur un évènement **e1** l'arrivée d'une ou plusieurs requêtes à arbitrer. Une fois la requête dans la liste, l'arbitre est réveillé par la notification de **e1** de la présence de requêtes à traiter. Le processus du module utilisateur se met en attente d'un évènement **e2** quand sa demande a été déposée.

3. l'arbitre choisit le module selon la politique d'arbitrage (FIFO, plus haute priorité, LRU). Dans l'exemple, **R2** est choisit suite à un arbitrage de plus haute priorité. L'arbitre réveille le module utilisateur par une notification de **e2**.

4. la lecture continue en appelant la méthode *Read()* implémentée par la mémoire. Cette méthode effectue une lecture d'une donnée à l'adresse indiquée puis retourne cette donnée au module.

Le même principe s'applique pour une écriture vers un périphérique, fonction *Write()* du canal, ou pour un acquittement d'une écriture bloquante, fonction *CommandOut()*.

Calcul de la latence de communication :

Le calcul de la latence de communication s'effectue dans les fonctions *Read()*, *Write()* ou *CommandOut()*. La fonction *wait()* est utilisée pour simuler la latence. Elle permet d'interrompre un processus et d'appeler l'ordonnanceur SystemC pour que celui-ci donne la main à un autre processus dans la simulation. Dans les fonctions, le calcul de la latence s'effectue à partir des équations suivantes :

$$\begin{aligned} \text{Mode rafale : } L_C &= N * (L_{AR} + L_{AD/T} + L_{T/AC}) \\ \text{Mode non rafale : } L_C &= L_{AR} + N * (L_{AD/T} + L_{T/AC}) \end{aligned} \tag{3.1}$$

Avec **Lc** : latence de communication, **N** : nombre d'accès sur le canal (fonction de la taille du bus), **Lar** : latence configurable liée à l'arbitrage, **Lad/t**: latence configurable liée au décodage d'adresse et **Lt/ac** : latence configurable liée au transfert d'une donnée et à l'acquittement.

3.3.2 Le pont fonctionnel

3.3.2.1 Caractéristiques

Un pont est utilisé pour relier deux bus. Les bus peuvent êtres différents ou identiques et peuvent donc travailler indépendamment. À partir de la documentation technique [84][85][81][4] de plusieurs ponts des protocoles AMBA et CoreConnect, les caractéristiques suivantes d'un pont ont pu êtres identifiées :

- Un pont possède une interface esclave d'un côté pour recevoir des requêtes du bus initiateur et une interface maître pour former une requête sur le bus
- Dans des bus hiérarchiques, le pont sépare un bus système (haute performance) d'un bus de périphérique (faible performance, puissance dissipée faible). Ceci évite la surcharge du bus système avec plein de périphériques et laisse la bande passante aux composants rapides (processeur, DMA, etc).
- Le pont permet d'adapter deux domaines d'horloge différents. Des rapports de fréquence entre le bus rapide et lent synchrones de 1:1, 2:1, 3:1 et 4:1 peuvent être possibles. La fréquence du bus lent doit être inférieure ou égale à celle du bus rapide (e.g. dans un rapport 2:1, si le bus PLB tourne à 200 MHz alors le bus OPB roule à 100 MHz).
- Dans des rapports de fréquence 2:1, 3:1 et 4:1, le pont nécessite une **zone tampon** d'une certaine profondeur en mots (32 bits) pour permettre l'adaptation

des deux domaines d'horloge et éviter de ralentir le bus système. De plus, la présence de cette zone tampon permet de supporter le mode rafale à condition que les deux bus (système et périphérique) offre le mode rafale. C'est le cas du pont PLB-OPB, OPB-PLB. La zone tampon permet aussi d'adapter la taille des deux bus (e.g. bus de données PLB de 64 bits et bus de données OPB de 32 bits). Des registres sont parfois présents pour synchroniser les horloges, ce qui peut rajouter des latences supplémentaires dans le transfert des signaux.

- Dans des rapports de fréquence 1:1 et si les bus sont de même taille (cas des bus AHB, APB et OPB), le pont ne requiert pas de zone tampon. Ces ponts sont **directs** (pont AHB-APB et OPB-OPB).

- Un pont est par défaut **unidirectionnel**. Il faut deux ponts pour une bidirectionnalité.

Ces caractéristiques se retrouvent dans le fonctionnement du pont fonctionnel. Ce dernier est configurable à l'aide des paramètres suivants :

- Pont Direct ou Stocke-et-Transfert (i.e. pont avec zone tampon)

- Latence de transfert d'un bus à l'autre

- Latence supplémentaire au besoin (e.g. acquittement Space)

- Nombre de fenêtres d'adresse (i.e. le pont donne accès à un certain nombre de composants qui ont une plage adresse sur le bus destinataire)

- Schéma de latence pour le mode Stocke-et-Transfert, i.e. pont PLB-OPB ou OPB-PLB

- Deux périodes d'horloge liés au fréquence de fonctionnement des deux canaux TF. En mode Direct, les deux périodes sont identiques.

3.3.2.2 Fonctionnement interne

La figure 3.7 montre le diagramme de classe du pont fonctionnel en gris foncé ainsi que sa représentation SystemC.

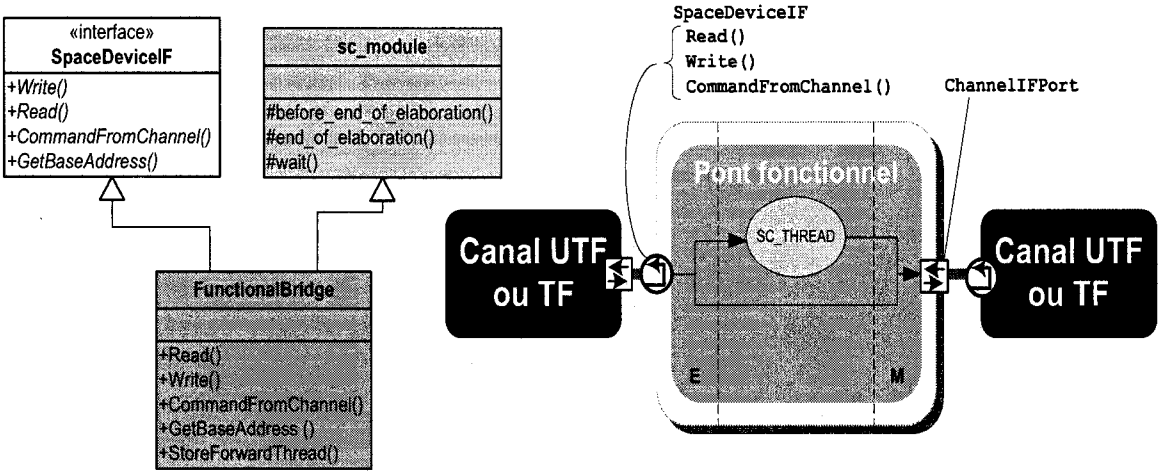

Figure 3.7 Diagramme de classe du pont fonctionnel

Le pont fonctionnel dérive de la classe SystemC, *sc_module*, et de l'interface *SpaceDeviceIF*. Il implémente les fonctions de communication *Read()*, *Write()* et *CommandFromChannel()* de l'interface. Le port *ChannelIFPort* permet de se connecter à un canal UTF ou TF. Il possède une interface esclave (**E**) et maître (**M**). En mode direct, le pont envoie directement la transaction sur l'autre canal. Alors qu'en mode Store-et-Transfert, un processus *StoreForwardThread()* est instancié pour reproduire l'adaptation de deux domaines d'horloge différents.

Déroulement d'une communication :

L'exemple 3.8 d'une écriture d'un module dans une mémoire illustre le principe de déroulement d'une communication à travers le pont fonctionnel.

Dans le mode **Direct**, le pont ne fait que transférer la transaction d'un canal à un autre en simulant une latence de transfert calculée en fonction du nombre de données

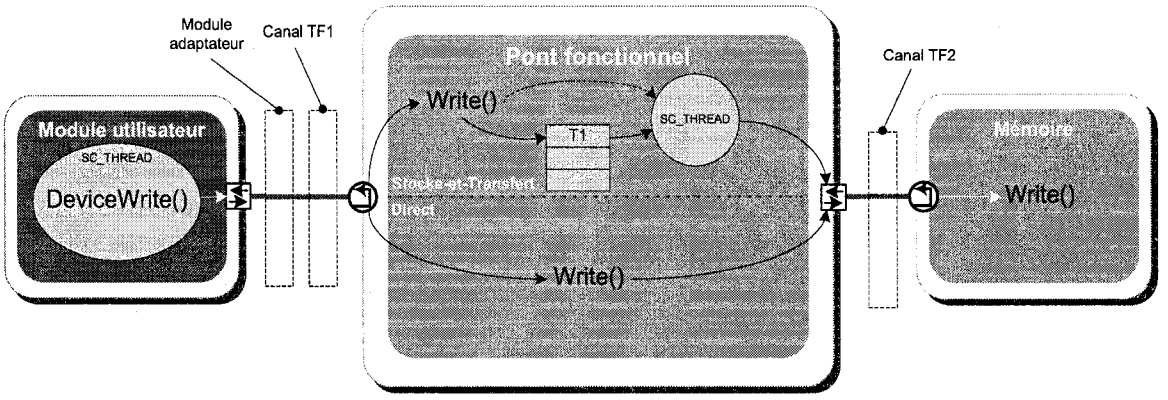

Figure 3.8 Communication à travers le pont fonctionnel

contenus dans la transaction. Soit le module utilisateur initiant une transaction composée de 4 mots (4*32 bits). La transaction passe par le canal TF1 qui simule le transfert de ces 4 mots avec une des formules 3.1. Par la suite, la transaction arrive dans le pont en appelant la méthode *Write()*. Dans cette méthode, une latence de transfert de la transaction est simulée. La documentation technique des ponts AHB-APB et OPB-OPB montre que les signaux mettent **un cycle** pour être présentés au bus suivant. Par conséquent, la latence de transfert associée à la transaction de 4 mots dans l'exemple sera de 4 cycles. La transaction se retrouve sur le canal TF2. A nouveau, une latence de transfert est simulée. La mémoire effectue l'écriture des données de la transaction. Enfin, la fonction retourne. C'est donc une simple séquence d'appel de fonctions avec quelques *wait* pour simuler des latences.

Dans le mode **Stocke-et-Transfert**, le pont utilise une FIFO pour stocker les transactions et un processus pour adapter deux domaines d'horloge différents pour les canaux TF. Soit le module utilisateur initiant une transaction composée de 4 mots (4*32 bits). De la même façon qu'en mode Direct, le canal TF1 simule une certaine latence de transfert par rapport au nombre de données compris dans la transaction. La transaction arrive dans le pont fonctionnel par la méthode *Write()*. Elle est alors stockée dans une FIFO puis un événement est notifié pour réveiller le processus du pont. En fonction du schéma de latence (pont PLB-OPB, OPB-PLB), une latence

peut être simulée à l'écriture et/ou à la lecture de la transaction dans la FIFO. Cette latence simulée est due à des interfaces pipelinées dans les vrais ponts. Il n'est pas possible de personnaliser les latences, il faut utiliser un schéma disponible. Dans la fonction *Write()*, le processus du module continue son exécution une fois la transaction écrite dans la FIFO. La transaction arrive sur le canal TF2 qui simule une latence de transfert. Puis les données de la transaction sont écrites en mémoire. La présence du processus et de la FIFO permet de découpler les deux domaines d'horloge en simulation. Ainsi, le module peut renvoyer une transaction à travers le pont alors que la transaction précédente est acheminée à la fréquence du deuxième canal. Si la FIFO simulée est pleine alors le processus du module initiateur attend sur un événement que la FIFO se vide. Dans la réalité, une FIFO asynchrone permet de réaliser ce découplage.

Pour une lecture en mode Stocke-et-Transfert, il n'y a pas de FIFO ni de processus car la transaction n'est complète qu'une fois les données lues. Un module ne peut pas lancer une deuxième lecture alors que la première n'est pas finie. Dans la réalité, il existe une FIFO qui sert à accumuler les lectures faites sur le deuxième bus pour optimiser les communications. En simulation, ceci n'est pas nécessaire.

3.4 Les composants au niveau BCA

Ce niveau transactionnel permet de raffiner les détails de communication. Les différentes phases de requêtes sur un bus sont reproduites (arbitrage, transfert, acquittement) avec des latences précises. Le concepteur dispose donc d'une meilleure estimation des temps de communication. La simulation reste rapide et permet de se focaliser sur les performances de l'application liée au réseau de communication choisi. Le " monitoring " permet toujours d'analyser des résultats post-simulation (utilisation du bus, nombres d'accès mémoires pour chaque module, communications entre chaque module). La figure III.2 de l'annexe III décrit l'ensemble des composants

du niveau BCA.

3.4.1 Bus OPB

3.4.1.1 Caractéristiques

Tel que mentionné, le bus OPB (*On-chip Peripheral Bus*) est employé pour relier des périphériques lents (UART, port Ethernet, contrôleur d'interruption, etc) déchargeant ainsi le bus système. Xilinx a développé le processeur logiciel, MicroBlaze [86], qui s'interface avec le bus OPB. Ceci permet la création d'un système monoprocesseur sans l'utilisation d'un bus système avec un processeur matériel comme le PowerPC. Des systèmes multiprocesseurs à base de processeurs MicroBlaze et de bus OPB sont aussi réalisables avec des performances satisfaisantes [67][51][28].

Le modèle abstrait modélise certaines fonctionnalités décrites dans les spécifications techniques du bus OPB [83][30]. Il repose sur un modèle de communication multi-maîtres (initiateur de requêtes) – multi-esclaves (répondant aux requêtes) faisant intervenir un arbitre pour l'attribution du bus à un maître à la fois. Le modèle abstrait implémente les caractéristiques suivantes :

- Bus synchrone de données et d'adresse de 32 bits.
- Transferts d'octets (alignés ou non), de demi-mots et de mots.
- Verrouillage du bus (un maître indique à l'arbitre qu'il verrouille le bus empêchant un autre maître d'avoir le bus).
- Mode séquentiel (permet d'économiser des cycles d'arbitrage pour le maître, l'esclave incrémente l'adresse, toujours associé à un verrouillage du bus).
- **Arbitre** avec 3 politiques d'arbitrage (FIFO, plus haute priorité, LRU) et un

mécanisme de délai d'attente (***timeout***) pour libérer le bus en cas de blocage d'une requête.

- Possibilité pour l'esclave de supprimer le délai d'attente (*timeout*) de l'arbitre.
- Support du *Byte Enable* permettant des accès rapides à des données contiges non alignées (e.g. un accès de 3 octets). Le maître et l'esclave doivent supporter cette option pour l'utiliser.

Le présent travail a porté sur l'arbitre et la gestion du timeout (en gras souligné dans la liste). Les autres caractéristiques étaient déjà présentes [23]. L'annexe V présente une comparaison des caractéristiques disponibles entre le modèle abstrait et le bus OPB implémenté par Xilinx selon le protocole d'IBM.

Dans le modèle abstrait, le maître et l'esclave suivent le schéma : **requête, attribution, transfert, acquittement**. Un maître OPB effectue ainsi les étapes suivantes :

1. séparer la transaction en plusieurs transferts et la préparer (entête au besoin),
2. demander le bus,
3. fixer les options de transferts (e.g. mode verrouillé, séquentiel),
4. émettre le ou les transferts,
5. et traiter l'acquittement (réussite ou erreur).

L'esclave, quand à lui, effectue les opérations suivantes :

1. inhiber le délai d'attente de l'arbitre au besoin,
2. lire ou écrire la donnée,
3. acquittement en tenant compte de la réussite ou de l'échec (*timeout*, erreur d'écriture/lecture) lors du traitement de la requête.

3.4.1.2 Fonctionnement interne

La figure 3.9 montre le diagramme de classe du bus OPB en gris foncé ainsi que sa représentation SystemC.

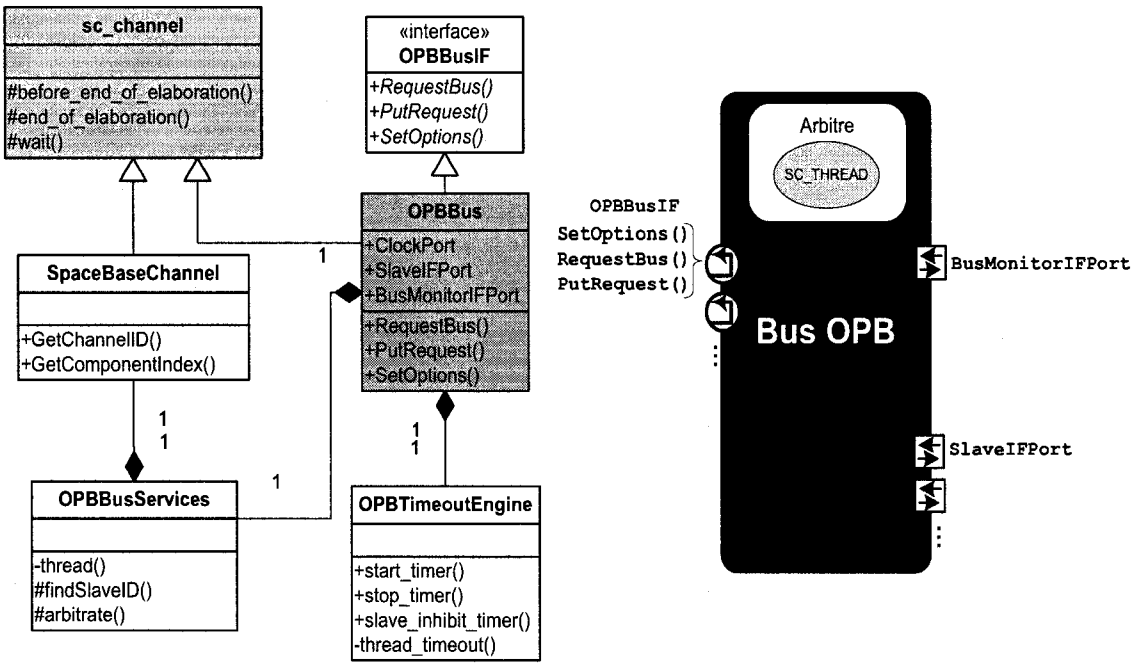

Figure 3.9 Diagramme de classe du bus OPB

Le bus OPB appelé *OPBBus* dérive de *OPBBusIF* et de la classe de base SystemC, *sc_channel*. L'interface *OPBBusIF* fournit les méthodes *RequestBus()*, *SetOptions()* et *PutRequest()* qui sont appelées par l'OPB adaptateur. Respectivement, ces méthodes permettent de demander le bus, de fixer les options de transferts et d'envoyer la requête d'écriture ou de lecture sur le bus. Le bus OPB se compose d'un arbitre, d'un temporisateur (*OPBTimeoutEngine*) et de deux ports, *SlaveIFPort* et *BusMonitorIFPort*. La fonctionnalité de l'arbitre est assuré par *OPBBusServices* grâce au processus *thread()* et à la méthode *arbitrate()*. Le temporisateur s'occupe de compter 16 cycles à chaque fois qu'un transfert passe sur le bus. L'esclave doit répondre au maitre en moins de 16 cycles. Le port *SlaveIFPort* permet au bus OPB

d'accéder aux méthodes *SlaveRead()* et *SlaveWrite()* implémenté par un périphérique ou un OPB adaptateur et le port *BusMonitorIFPort* sert au *monitoring* pour des statistiques sur les communications.

Déroulement d'une communication :

L'exemple 3.10 d'une lecture d'un module dans une mémoire illustre le principe de déroulement d'une communication à travers le bus OPB.

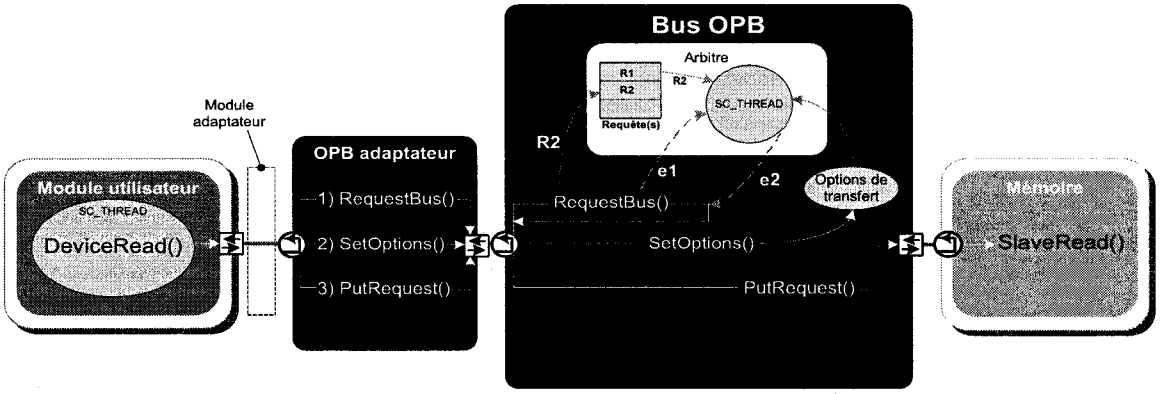

Figure 3.10 Communication sur le bus OPB

Le module utilisateur effectue un *DeviceRead()*. Cette méthode suit une séquence d'appel de fonctions à travers les interfaces qui la mène à l'OPB adaptateur. Dans ce dernier, les phases du transfert s'effectuent :

1. *RequestBus()* : l'accès au bus est demandé à l'arbitre. La requête **R2** est mise dans une liste et l'arbitre choisit un module selon une politique d'arbitrage (e.g. plus haute priorité). Le processus de l'arbitre est réveillé par notification d'un événement, ici **e1**, et celui-ci réveille le processus du module aussi par une notification d'un événement, **e2**. L'arbitre simule un délai de 2 cycles pour arbitrer [83] par un *wait()*.

2. *SetOptions()* : une fois le bus accordé, le module ou maitre envoie toutes les options de contrôle pour le transfert. Par exemple, requête de lecture de

données 32 bits en mode séquentiel et verrouillé. Certaines options de transfert influencent l'arbitre. En effet, si le bus était verrouillé et qu'un autre module demande le bus, l'arbitre attend que le bus se libère avant d'arbitrer.

3. *PutRequest()* : le transfert est alors envoyé sur le bus OPB. Un délai de un cycle est simulé pour cet accès. La méthode *SlaveRead()* est alors appelée. L'opération de lecture est effectuée par la mémoire qui renvoie un acquittement lors du retour de la méthode. Un *wait()* de un cycle dans la mémoire simule l'acquittement et l'envoi de la donnée lue.

Le même principe s'applique pour des écritures vers des périphériques (UART, mémoire, minuterie, etc) et vers des modules. La lecture vers des modules est locale et s'effectue dans le module adaptateur (cf. partie 3.1.1).

Description des politiques d'arbitrage implémentées dans l'arbitre :

L'arbitre du bus OPB implémente trois politiques d'arbitrage :

- FIFO (*First In First Out*) : les demandes d'accès au bus arrive dans un certain ordre. L'arbitre attribue le bus au maître au premier de la liste des demandes d'accès au bus. Ce maître est donc le premier à avoir demandé le bus. Toute nouvelle demande d'accès est placée à la fin de la liste.

- Plus haute priorité : l'arbitre attribue le bus au maître ayant la plus haute priorité. Dans Space, chaque maître se voit attribuer un ID qui devient sa priorité. L'ID le plus faible est la plus haute priorité. Un processeur a par défaut la priorité la plus élevée.

- LRU (*Least Recently Used*) : cet algorithme est considéré comme dynamique par la documentation de Xilinx car les priorités des maîtres évoluent en fonction de l'attribution du bus OPB. Le principe du LRU est l'attribution du bus au maître ayant le moins obtenu le bus dans le passé récent. Le maître le moins

prioritaire est celui qui s'est fait attribué le bus le plus récemment. Sur le FPGA, le bus OPB supporte 16 maîtres donc 16 niveaux de priorité possibles. Le modèle abstrait supporte 255 modules au maximum donc 255 niveaux de priorités si le système possédait que des maîtres or il y a forcément des esclaves. La figure 3.19 présente l'algorithme LRU implémenté dans l'arbitre. Il consiste à gérer deux listes : la liste des demandes d'accès et la liste des IDs de maîtres où la position dans la liste détermine la priorité du maître. Au départ, la liste des IDs est vide donc le premier accès sur le bus initialisera la liste. Sur l'organigramme, cela correspond à la zone 1. Par la suite, la zone 2 permet de traiter les demandes d'accès normalement. Si parmi les demandes d'accès, un maître n'a jamais eu le bus, il aura le bus et son ID sera sauvegardé à la fin de la liste des IDs et il devient le moins prioritaire. Sinon le maître le plus prioritaire sera cherché dans la liste des IDs. Ceci conduit à deux fins possibles dans la zone 2 (partie en gris foncé).

Gestion du *timeout* :

La figure 3.11 montre le fonctionnement du temporisateur dans le bus OPB.

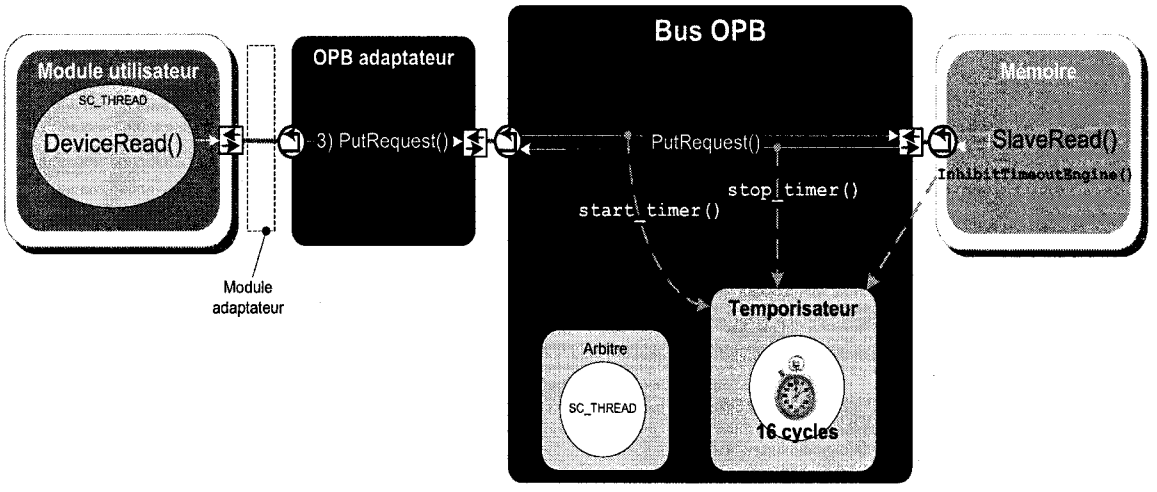

Figure 3.11 Timeout dans la bus OPB

Lorsque le transfert de la donnée est effectué sur le bus par l'appel à la méthode *PutRe-*

quest(), le processus du temporisateur est réveillé par une notification d'évènement, méthode *start_timer()*. Le processus se met en multiples attentes dans un *wait()* : attente de 16 cycles, attente d'un évènement indiquant la fin du transfert ou attente d'un évènement pour inhiber le temporisateur. Si le transfert se passe normalement et que l'esclave acquitte le transfert avant le $16^{ème}$ cycle, le décompte du temporisateur est arrêté par *stop_timer()*. Le processus retourne alors en attente d'un début de transfert. Si 16 cycles se sont écoulés avant l'acquittement de l'esclave, le processus se réveille et indique un *timeout*. Au retour dans l'OPB adaptateur, l'acquittement est examiné et le transfert sera réémis en cas de *timeout*. Pour des esclaves avec une grande latence où l'acquittement peut arriver après 16 cycles, ce dernier peut inhiber le décompte par l'intermédiaire de *InhibitTimeoutEngine()* à l'arrivée de la donnée, traiter la donnée et réactiver le temporisateur lors de l'acquittement. Cette option est utilisée dans le pont OPB-OPB. Pour les autres esclaves, le temporisateur est actif. Dans le chapitre des résultats, nous verrons l'impact de la gestion du *timeout* dans un système multibus.

3.4.2 Pont OPB-OPB

3.4.2.1 Caractéristiques

Le pont OPB-OPB est utilisé pour relier deux bus OPB. En fonction de la documentation technique de Xilinx [84], le modèle haut niveau en SystemC offre les caractéristiques suivantes :

- Il possède une interface esclave d'un côté pour recevoir des requêtes du bus OPB initiateur et une interface maître pour former une requête sur le bus OPB destinataire.

- Il ne permet pas d'adapter deux domaines d'horloge différents. Par conséquent,

les deux bus OPB doivent travailler à la même fréquence et avoir la même largeur.

- Il n'a pas de zone tampon (*buffer*). Il ne supporte donc pas le mode rafale. Le pont OPB-OPB est **direct**.
- Il est **unidirectionnel**. Il faut deux ponts OPB-OPB pour une bidirectionnalité.
- La latence de transfert d'un bus à un autre est de **un cycle**.
- Le nombre de fenêtres d'adresse est configurable (i.e. le pont donne accès à un certain nombre de composants qui ont une plage adresse sur le bus destinataire).
- Il y a un mécanisme de détection d'interblocage (mode *deadlock*) que l'on peut activer ou pas. Si ce mécanisme est désactivé, c'est un mécanisme d'information de famine (mode *starvation*) qui est actif. Le mode anti-interblocage est aussi offert pour casser un interblocage pour une paire de ponts OPB-OPB.
- Il assure l'atomicité des transactions Space lors d'une communication entre un module logiciel et matériel.

3.4.2.2 Fonctionnement interne

La figure 3.12 montre le diagramme de classe du pont OPB-OPB en gris foncé ainsi que sa représentation SystemC.

Le pont OPB-OPB dérive de la classe *OPBSlave* et de l'interface *DeadlockBridgeIF*. Il implémente les fonctions de communication *SlaveSpecificRead()*, *SlaveSpecificWrite()* et *CheckDeadlock()* qui permet de détecter les interblocages. Le port *BusIFPort* permet de se connecter à un bus OPB. Il possède une interface esclave représentée par l'héritage de *OPBSlave* et une interface maître représenté par la composition

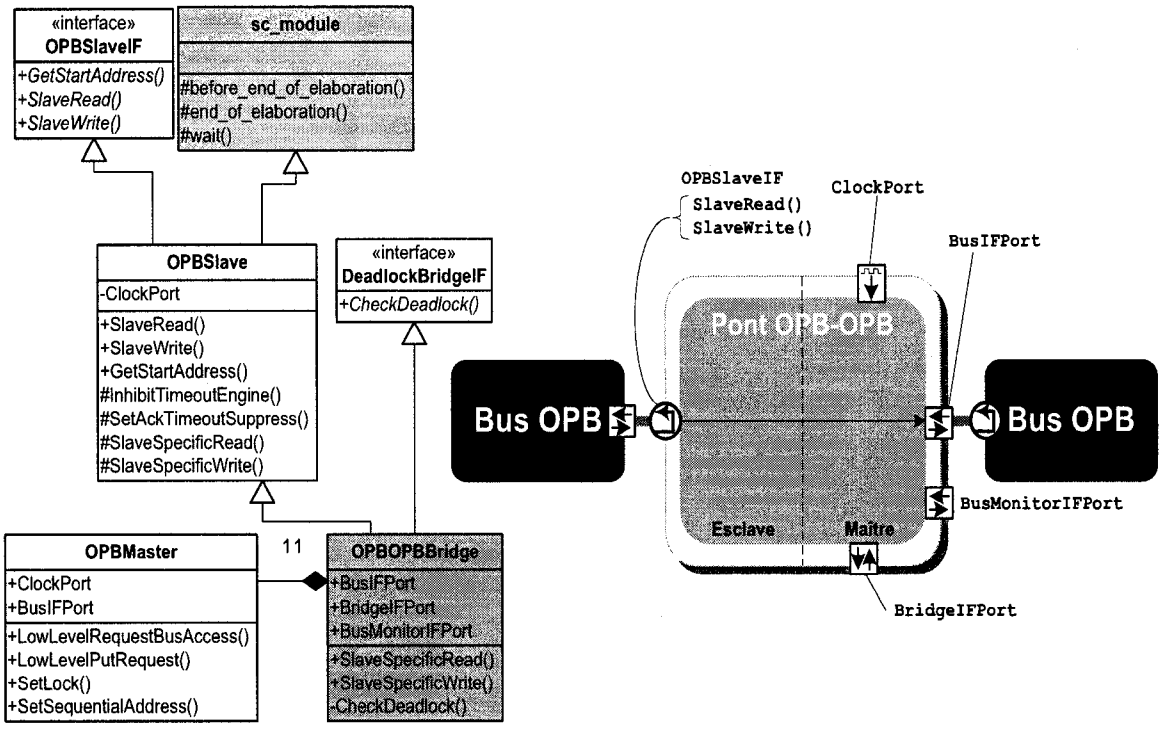

Figure 3.12 Diagramme de classe du pont OPB-OPB

de la classe *OPBMaster*. Le port *BusMonitorIFPort*, *BridgeIFPort* et *ClockPort* permettent respectivement d'obtenir des statistiques sur les requêtes qui transitent par le pont, de se connecter à un autre pont OPB-OPB pour détecter les interblocages et de recevoir l'horloge du système.

Déroulement d'une communication :

L'exemple 3.13 d'une écriture d'un module dans une mémoire illustre le principe de déroulement d'une communication à travers le pont OPB-OPB.

Lorsque le module utilisateur veut écrire vers une mémoire qui se trouve sur un autre bus OPB, il passe par le pont OPB-OPB. La requête arrive dans l'interface esclave du pont par l'intermédiaire de la méthode *SlaveWrite()*. La fonctionnalité du pont est implémenté dans *SlaveSpecificWrite()*. En passant par le bus OPB1, le temporisateur de ce bus s'est déclenché, l'interface maître du pont a 16 cycles pour

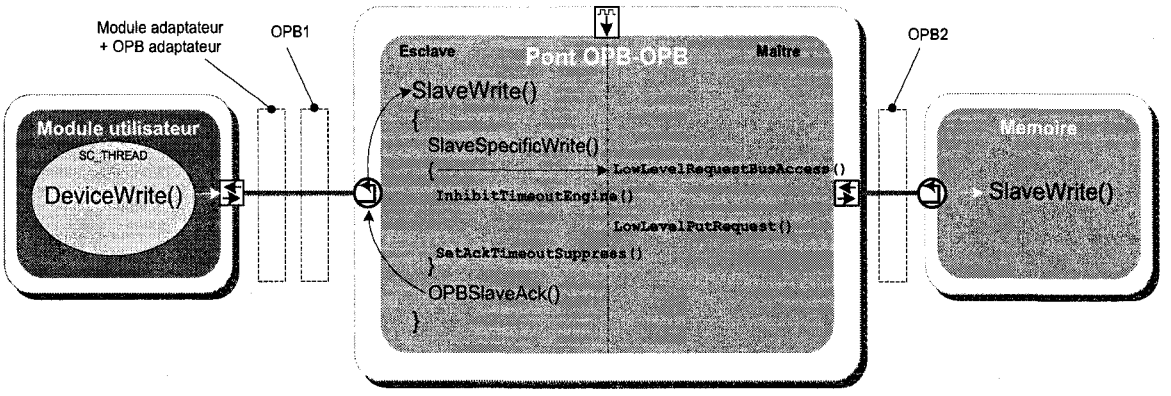

Figure 3.13 Communication dans le pont OPB-OPB

obtenir accès au bus OPB2 sinon la requête est interrompue et l'OPB adaptateur doit réémettre la requête après un arbitrage. L'interface maitre demande le bus OPB2 par *LowLevelBusAccess()*. Il redemande le bus jusqu'à ce qu'il ait obtenu. À chaque tentative, un cycle d'attente est simulé. Quand le bus lui est accordé, l'interface esclave inhibe le temporisateur du bus OPB1, *InhibitTimeoutEngine()*. Un cycle de transfert d'un bus à un autre est simulé par un *wait()* sur un front montant d'horloge. Puis la requête est envoyée sur le bus OPB2. Le temporisateur de ce bus part. L'esclave, ici la mémoire a 16 cycles pour traiter la requête et acquitter. Une fois le traitement effectué par l'esclave, nous revenons dans le pont côté maître. Si un *timeout* ou une erreur d'acquittement a eu lieu, la requête est réémise sur le bus OPB2. Sinon, l'interface esclave réactive le temporisateur du bus OPB1 pour la prochaine requête, méthode SetAck*TimeoutSuppress()*. Puis un acquittement du pont, *OPBSlaveAck()*, est opérée. Ici, une latence d'acquittement de un cycle est simulée.

Détection d'interblocage et de famine :

La figure 3.14 présente le mécanisme qui permet de détecter un interblocage (*deadlock*) entre deux bus OPB et d'informer le concepteur de risque important de famine (*starvation*).

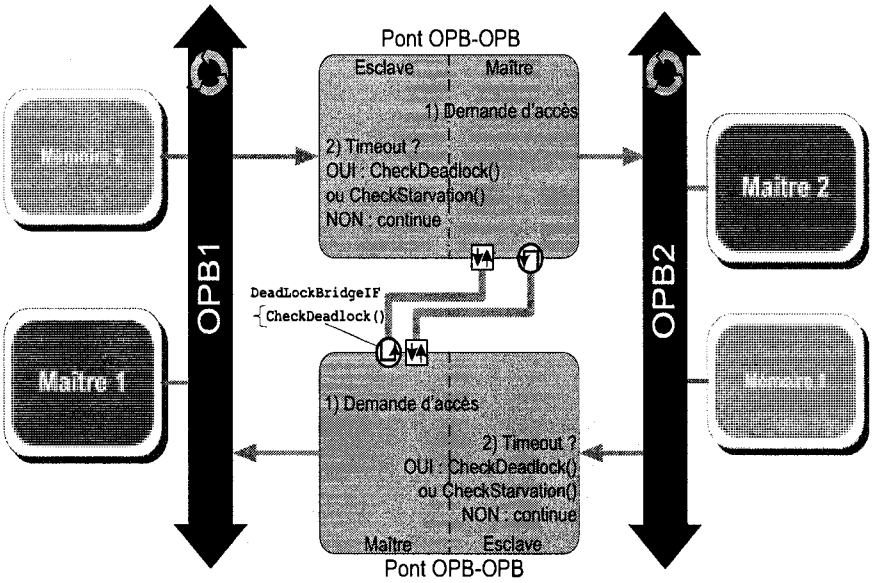

Figure 3.14 Gestion du timeout dans le pont OPB-OPB

Un **interblocage** se produit lorsque deux maîtres (Maître 1 et 2 sur le schéma 3.14) sur deux bus OPB différents communiquent avec un autre maître ou un périphérique (Mémoire 1 et 2 sur le schéma) à travers un pont OPB-OPB. Ils demandent l'accès à leur bus respectif (OPB1 et OPB2) et l'obtiennent au même moment. La requête arrive dans les ponts OPB-OPB. Chaque interface maître des ponts demandent le bus OPB mais ce dernier est pris par les maîtres 1 ou 2. Donc une possible situation d'interblocage se produit. Grâce au *timeout* des bus OPB1 et OPB2, les requêtes de chaque maître sont interrompues mais il se peut qu'au réarbitrage sur les deux bus, la situation se reproduise indéfiniment ou jusqu'à ce qu'un autre maître, autre que ceux impliqués dans l'interblocage, prenne la main d'un bus permettant ainsi de casser l'interblocage. En simulation, un mécanisme permet de détecter un interblocage et avertir le concepteur du risque dans son système. Le mécanisme agit comme suit :

- Les deux ponts OPB-OPB sont reliés ensemble.
- L'interface maître du pont OPB-OPB demande l'accès au bus (1). L'arbitre du bus OPB répond par un accès non attribué car le bus est déjà pris par un maître.

- L'interface maître vérifie si un *timeout* (16 cycles écoulés) a eu lieu coté esclave (2). Si non, le pont redemande l'accès au bus OPB (1) après une latence de un cycle. Les étapes (1) et (2) sont donc répétées jusqu'à ce que le pont obtienne le bus. Si oui, cela veut dire que le pont s'est fait refusé 16 fois l'accès au bus. Le pont OPB-OPB appelle la méthode *CheckDeadlock()* implémenté dans l'autre pont OPB-OPB. Un compteur comptabilise le nombre de *timeout* ayant eu lieu pour un maître donné (e.g. Maître 1). La valeur du compteur de *timeout* est passée en paramètre. Cette valeur sera comparée à celle du compteur de l'autre pont OPB-OPB. Si les deux valeurs sont égales, un interblocage est détecté et la simulation est arrêtée. En effet, un interblocage dans le cas spécifique du bus OPB avec l'utilisation du temporisateur conduit souvent à la même séquence d'arbitrage des deux maîtres et ces derniers arrivent au même moment dans les ponts.

La méthode *CheckDeadlock()* est appelée uniquement si le mode *deadlock* est actif et s'il y a eu par exemple N *timeouts* voulant dire N fois la même séquence où les ponts n'obtiennent jamais le bus OPB. Le nombre N est configurable à l'instanciation du pont. Par défaut, il vaut 3, i.e. trois timeouts consécutifs pour le même maître soient 3*16 = 48 accès non attribués pour le pont.

Alors que l'interblocage se produit à la suite de communications inter-bus, le phénomène de **famine** (*starvation*) peut se produire à cause des communications intra-bus et de la priorité des maîtres. Imaginons que le Maître 1 et 2 veulent écrire 64 données de 32 bits dans la mémoire 1, que l'arbitre du bus OPB2 utilise un arbitrage de plus haute priorité et que Maître 2 soit plus prioritaire que Maître 1. Le Maître 1 passe par le pont OPB-OPB et le Maître 2 reste sur le bus OPB2. La famine consiste en ce qu'un maître moins prioritaire obtienne très rarement le bus à cause d'un maitre plus prioritaire. E.g. Maître 1 n'a pas souvent le bus OPB2 car Maître2 est plus prioritaire. Dans ce cas, l'interface maitre du pont OPB-OPB a

souvent un accès non attribué. En mode *starvation*, lorsqu'un *timeout* se produit, la méthode *CheckStarvation()* permet d'informer d'une possible famine. Cette fois-ci, la valeur du compteur est comparée à une valeur X (configurable à l'instanciation) représentant un certain nombre de *timeouts*. Si le compteur atteint la valeur X, cela veut dire qu'un maître n'a pas eu X*16 fois le bus OPB consécutivement. Par défaut, X vaut 10 soit 160 accès non attribués successivement. Une solution simple est de changer l'arbitrage de bus OPB2 pour un **arbitrage LRU** permettant ainsi au Maître 1 d'avoir le bus aussi souvent que Maître 2.

Atomicité des communications Space :

Lors d'une communication entre modules, un message Space est envoyé avec un entête de 32 bits permettant au module destinataire de reconstituer le message scindé en plusieurs transferts sur le bus OPB. L'option de verrouillage du bus OPB (*bus lock*) est utilisée pour assurer l'atomicité du message Space, i.e. l'envoi du message ne peut être interrompu par un autre maître. Dans le cas d'un message transitant à travers le pont, l'atomicité doit aussi être assurée sur le deuxième bus. Le protocole OPB [30] spécifie que lors de l'utilisation du signal de verrouillage, celui-ci doit être abaissé lors du transfert de la dernière donnée du message permettant un chevauchement de cycle d'arbitrage. Cette particularité est implémentée dans le modèle OPB abstrait et est utilisée dans le pont OPB-OPB.

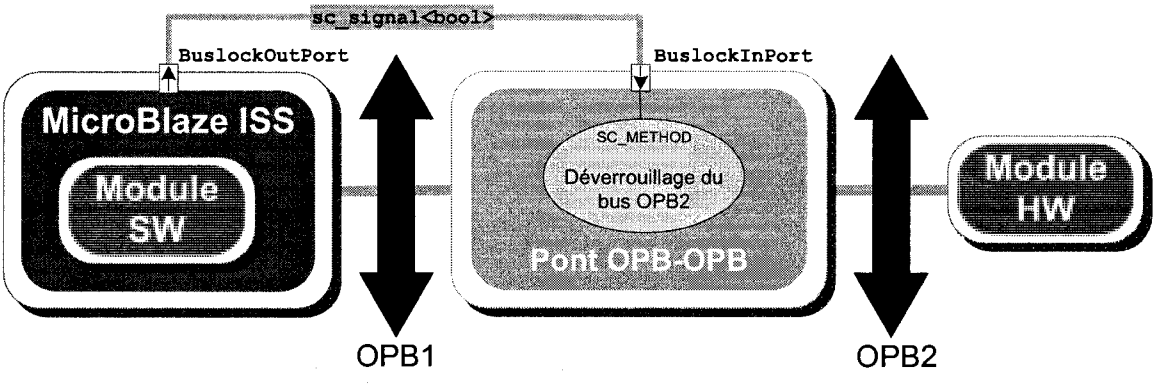

Figure 3.15 Atomicité des transactions Space à travers le pont OPB-OPB

Lors d'une communication entre modules matériels ou entre un module matériel et logiciel, le pont OPB-OPB détecte si le message est en mode verrouillage (les accès simples ne nécessitent pas de verrouillage). Si oui, l'interface maître du pont bloque le deuxième bus. Lorsque la dernière donnée du message arrive dans le pont, le verrouillage est désactivé, le pont le détecte et libère le deuxième bus.

En revanche, lors d'une communication entre un module logiciel et matériel, ceci n'est plus vrai. En effet, le processeur MicroBlaze permet de verrouiller/déverrouiller le bus OPB par programmation en écrivant un bit dans un registre spécial, le MSR (*Machine Status Register*). Lorsque le MicroBlaze initie un message sur le bus OPB, il déverrouille le bus 2 ou 3 cycles après l'envoi de la dernière donnée. En simulation, cela a pour conséquence que le pont OPB-OPB ne détecte pas la fin du verrouillage associé à un message. Le deuxième bus est alors bloqué indéfiniment et plus aucun message n'est arbitré. Pour remédier à ce problème en simulation, un signal de type booléen entre le processeur et les ponts OPB-OPB donnant accès à d'autres bus est utilisé (figure 3.15). Ainsi, lorsque le processeur débloque le bus OPB1 en écrivant dans le registre MSR, il écrit la valeur VRAI sur le signal booléen à travers le port *BuslockOutPort*. Le pont possède un processus SC_METHOD sensible sur les fronts montants du signal booléen. À la détection du front, le processus se déclenche et le bus OPB2 est libéré. Dans le cas où le MicroBlaze est connecté à un bus OPB sur lequel il y a plusieurs ponts OPB-OPB, le port *BuslockOutPort* est branché à tous les ports *BuslockInPort*. Chaque fois que le processeur initie un message sur OPB1 avec le verrouillage, que le message soit pour un module sur OPB1 ou OPB2, lors du déverrouillage de OPB1, le processus de chaque pont se déclenche. Mais cela n'a aucune conséquence car le déverrouillage de OPB2 n'a lieu que si un message avec la priorité du processeur existe dans le pont OPB-OPB.

3.5 Méthode des fenêtres dans les ponts

Soit deux bus, bus 1 et bus 2, dans un système multibus, une transaction envoyée par un module sur le bus 1 doit arriver au bon destinataire se trouvant sur le bus 2 en passant par un pont grâce à une adresse. Sur le FPGA de Xilinx, un pont OPB-OPB par exemple utilise des plages ou fenêtres d'adresse. Si l'adresse destinataire est détectée dans une des plages, la transaction est transférée sur l'autre bus. En simulation, au niveau TF comme au niveau BCA, une méthode des fenêtres similaire à celle du pont OPB-OPB a été implantée.

Le principe est qu'un pont donne accès aux composants sur l'autre bus via des fenêtres (figure 3.16). La méthode des fenêtres s'appuie sur les IDs des modules et des périphériques. Lors de l'instanciation de tous les composants du système à simuler, le constructeur du pont reçoit en paramètre un tableau contenant l'ID des composants auxquels ils donnent accès. À partir de l'ID, le pont est capable de constituer l'adresse de base des modules et périphériques pour former la table des fenêtres. Le pont aussi possède un ID qui permet de composer son adresse spécifique pour le reconnaître.

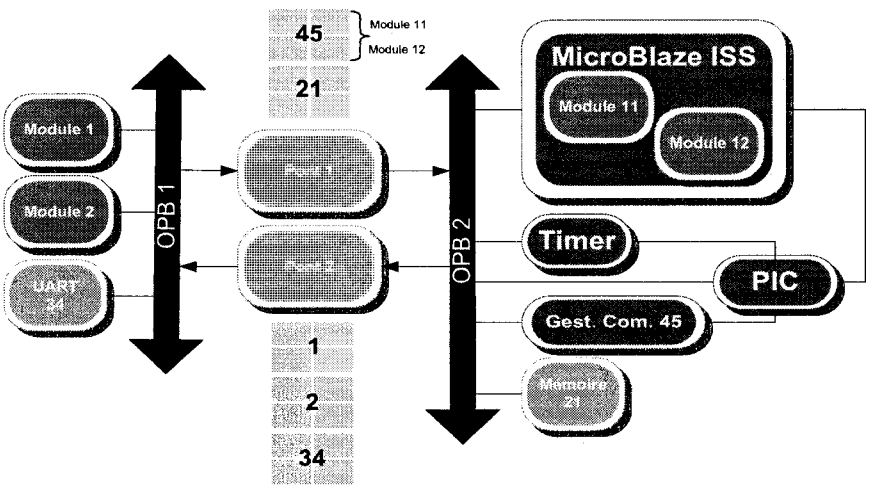

Figure 3.16 Méthode des fenêtres

Dans une simulation SystemC, un bus consulte une table interne pour savoir où diriger la transaction. L'index de cette table représente les IDs des composants connectés au bus. À un index n, se trouve l'index d'un autre tableau. Ce dernier est le tableau de ports qui regroupe tous les ports du bus auxquels sont connectés les composants. Ainsi, le bus retrouve le bon port pour appeler la méthode de communication du bon esclave. Pour constituer la table interne des composants du bus, celui-ci interroge tous les esclaves connectés lors de la phase d'élaboration de SystemC (méthode *end_of_elaboration()*). En interrogeant un composant esclave, celui-ci renvoie son adresse de base. Dans le cas du pont, ce dernier renvoie son adresse spéciale qui l'identifie. Le bus extrait l'ID du pont de cette adresse et a accès à la table des fenêtres associée à ce pont. De la sorte, il peut enregistrer tous les composants présents sur un autre bus dans sa table interne. Les composants sont comme attachés virtuellement au bus.

Considérons un exemple où le Module avec l'ID 2 désire écrire des données en mémoire identifiée par l'ID 21 :

1. la transaction arrive dans le bus OPB1. À partir de l'adresse de destination, OPB1 extrait l'ID 21 correspondant à la mémoire sur le bus OPB2.

2. OPB 1 regarde dans sa table interne de composants à l'index 21 et trouve le port à prendre dans le tableau de ports. Le port correspondant est celui du pont 1. Par conséquent, pour tous les composants accessibles depuis un autre bus situés sur OPB2 (Mémoire 21 et Gestionnaire de Communication 45), le port trouvé est toujours celui du pont 1. Même chose pour le bus OPB2 avec le pont 2.

3. la transaction passe par le pont 1 qui demande le bus OPB2, l'obtient puis transfère la transaction du Module 2 jusqu'à la mémoire.

Notez que dans la figure 3.16, lors d'une communication entre un module matériel et logiciel, les modules logiciels sont accessibles par le gestionnaire de communication

qui envoie une interruption à l'attention du processeur pour prévenir de l'arrivée de message. D'où son ID 45 qui se retrouve dans le pont 1.

La méthode des fenêtres offre donc les avantages suivants :

- Pas de limite pour le nombre de ponts par bus.
- L'utilisation de fenêtres s'apparente à l'utilisation des plages d'adresse dans le pont OPB-OPB en VHDL de Xilinx.

Il n'est pas possible aussi qu'une transaction traverse plusieurs profondeurs de bus avec la méthode des fenêtres. Cela impliquerait que le premier bus connaisse l'ID des modules et périphériques sur le troisième bus, ce que la méthode en place ne permet pas de faire. Sur le FPGA, les plages d'adresse des ponts OPB-OPB permettraient de le faire si le premier pont englobe une certaine plage d'adresse du deuxième pont. L'adresse de destination, une fois transférée sur le deuxième bus pourrait être détectée par le deuxième pont et transférée sur le dernier bus. Mais les applications où les données traversent plusieurs niveaux de bus font plutôt appel à des composants plus appropriés comme des routeurs dans un réseau-sur-puce.

3.6 Composants annexes pour aider à améliorer le réseau multibus

3.6.1 Lien point à point

Sur un FPGA, un lien point à point permet de relier directement deux unités de traitement, e.g. un processeur et un coprocesseur. Il n'a pas d'arbitrage et la latence est déterministe. La bande passante est élevée et permet notamment de désengorger un bus partagé en créant des chemins de données spécifiques. Dans Space, le principe de lien point à point [54, 26] est disponible pour les trois niveaux d'abstraction. Au niveau UTF et TF, cela permet de relier deux modules ensemble et au niveau BCA,

deux modules ou un processeur et un module. Les types de communication possibles sont présentés dans la figure 3.17.

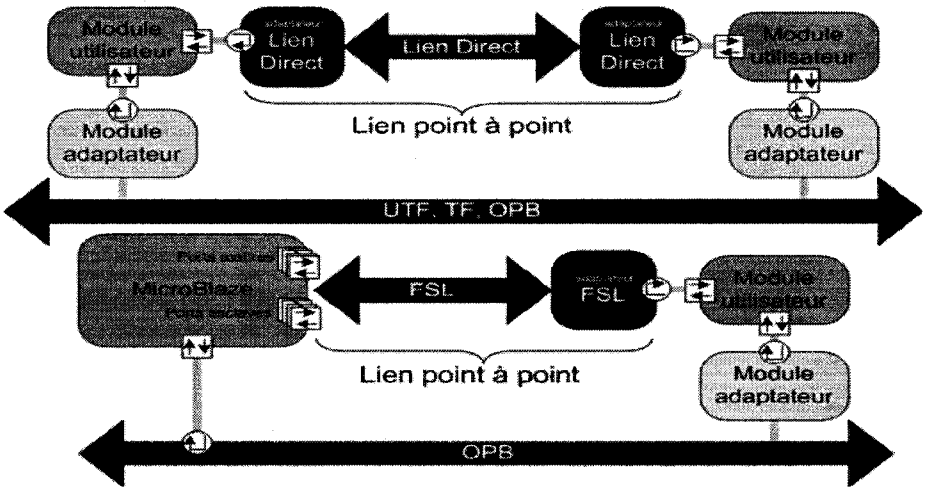

Figure 3.17 Lien point à point : types de communication

La communication entre deux modules reste transparente à l'utilisateur. Les fonctions de communication, *ModuleRead()* et *ModuleWrite()*, sont toujours employées. Un traitement spécial permet de router la transaction vers un lien point à point ou vers le bus. Dans le cas où un module destinataire est atteignable par un lien direct et par le bus, le lien point à point est prioritaire sur le bus puisque la communication est souvent plus rapide car il n'y a pas d'arbitrage.

Un lien point à point est unidirectionnel, il faut en instancier deux pour une bidirectionnalité entre deux modules. Un module utilisateur peut donc avoir un ou plusieurs liens point à point le reliant à un ou plusieurs modules.

Le stockage des données se fait à l'intérieur du lien point à point dans une FIFO. La taille de la FIFO est paramétrable à l'instanciation du lien. Les écritures et lectures bloquantes et non bloquantes sont supportées. Dans le cas d'une écriture non bloquante, quand la FIFO est pleine, après la dernière écriture ayant remplie la FIFO, le module bloque jusqu'à ce que le module destinataire est au moins lu une donnée. La latence de lecture et d'écriture est aussi paramétrable à l'instanciation.

Pour le lien FSL de Xilinx, dans le meilleur des cas, i.e. le module destinataire est prêt à recevoir la donnée, une écriture prend 2 cycles.

3.6.2 Mémoire BRAM double port

La librairie Space offre au niveau TF et BCA la possibilité d'utiliser des mémoires. Ces mémoires appelées *SpaceRAM* (niveau TF) et *OPBBRAM* (niveau BCA) permettent de reproduire des accès mémoires en écriture et en lecture. Les latences d'écriture et de lecture sont configurables pour représenter des mémoires avec un temps d'accès long (e.g. mémoire externe au FPGA comme de la SDRAM ou de la Flash) ou des mémoires rapides comme la BRAM sur le FPGA de Xilinx. Il est possible de spécifier la taille de la mémoire en octets comprise entre 64 Ko et 4 Mo en respectant des blocs de 64 Ko. La mémoire permet des accès d'octets, de demi-mots et de mots.

Dans un réseau multibus, une mémoire double port permet d'être accédée par deux modules en même temps se trouvant sur deux bus différents. Ce qui réduit les communications inter-bus à travers un pont, améliorant ainsi la bande passante des bus et le parallélisme du système.

Le modèle abstrait de la mémoire double port ne gère pas les écritures simultanées ni une écriture/lecture à la même adresse. En revanche, les lectures à la même adresse sont possibles. Il revient au concepteur d'écrire son application de telle sorte que deux modules accédant à la même mémoire n'écrivent pas au même instant dans le même emplacement. Même remarque pour une écriture/lecture.

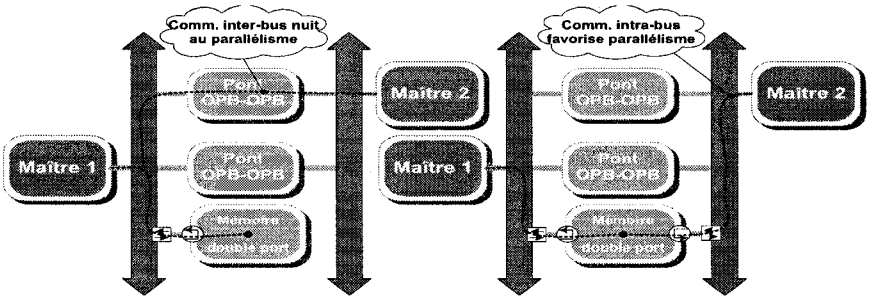

Figure 3.18 Mémoire double port

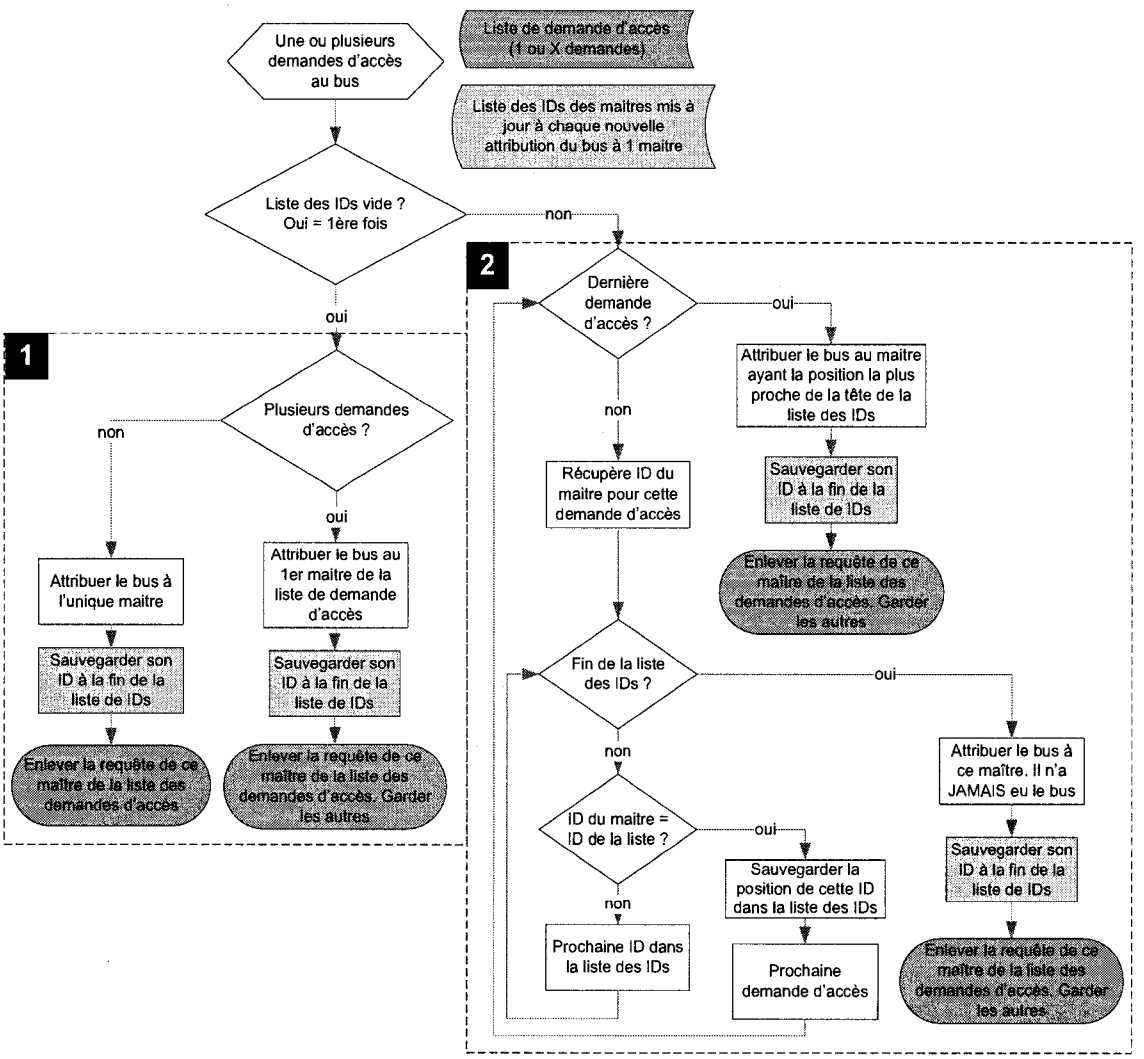

Figure 3.19 Algorithme LRU

CHAPITRE 4

ANALYSE DE L'EXPLORATION ET DES PERFORMANCES

Ce chapitre présente les performances obtenues à partir des composants décrits au chapitre 3. Tout d'abord les outils de mesure utilisés sont présentés. Par la suite, une comparaison des simulations au niveau TF et BCA[1] est présentée pour valider les modèles SystemC. Enfin, la performance de différentes topologies de bus est analysée à l'aide d'une application de décodage JPEG qui montre les risques liés à l'utilisation d'un pont et de différentcomms modes d'arbitrage.

4.1 Outils de mesure

Toutes les données montrées et analysées proviennent de bancs de test utilisés au niveau TF et BCA. Ces bancs de test ont permis d'évaluer les différents composants dans des systèmes simple bus ou multibus avec un ou plusieurs processeurs en travaillant sur le même niveau d'abstraction ou à différents niveaux. Les résultats ont été obtenus sur un ordinateur équipé d'un processeur **Intel Pentium 4 à 2,8GHz avec 1 Go de RAM sur lequel tourne Windows XP Pro Service pack 2**.

Tous les résultats présentés sont des résultats de simulation. Aucune implémentation physique sur un FPGA de Xilinx n'a été réalisée.

[1] Au niveau BCA, lors de la réalisation des tests, seul le bus OPB était disponible. Le bus PLB était en cours de développement.

4.1.1 La mesure du temps

Il faut distinguer deux types de temps lors d'une simulation SystemC : le **temps physique** et le **temps de simulation**.

Le temps physique représente le temps réel écoulé lors de l'exécution de la simulation. Ce temps peut s'exprimer en heures, en minutes, en secondes ou en millisecondes dépendamment de la complexité du système à simuler et du niveau d'abstraction. Il permet d'évaluer la performance d'un algorithme, d'une optimisation ou dans notre cas d'une simulation SystemC. Dans la librairie standard C/C++, certaines fonctions et macros contenues dans *time.h* ont permis d'obtenir et de manipuler le temps physique. Il s'agit de :

- **time_t *time*(time_t * timer)** : cette fonction retourne un objet de type *time_t* contenant le temps écoulé en seconde depuis la date du 1er janvier 1970 UTC.

- **clock_t *clock*(void)** : cette fonction retourne le nombre de tics d'horloge écoulés depuis le lancement d'un programme. Cette fonction est souvent associée à la macro CLOCKS_PER_SEC.

- **CLOCKS_PER_SEC** : cette macro permet d'obtenir le temps en seconde en spécifiant la relation entre le nombre de tics et la seconde. En divisant la valeur retournée par *clock()*, nous obtenons le nombre de secondes écoulés.

Le temps de simulation représente le temps d'exécution du système si celui-ci était implémenté sur son support réel (e.g. technologie FPGA, ASIC pour un système-sur-puce). Ce temps peut être qualifié de fictif ou virtuel par rapport au temps physique de la machine hôte de la simulation. La librairie SystemC offre quelques fonctions pour obtenir ce temps de simulation. Il s'agit de :

- **sc_simulation_time()** : cette fonction retourne une valeur de type *double* (nombre flottant en double précision sur 64 bits). Cette dernière représente le temps de simulation actuel dans l'unité de temps choisie. SystemC définit les unités de temps suivantes : SC_FS (femto), SC_PS (pico), SC_NS (nano), SC_US (micro), SC_MS (milli), SC_SEC (seconde). La plateforme Space utilise la nanoseconde (SC_NS) comme unité de temps par défaut.

- **sc_time_stamp()** : une autre fonction qui donne le temps actuel de la simulation. Elle retourne un object de type *sc_time*, une classe de SystemC qui permet de représenter des intervalles de temps et des temps de simulation.

4.1.2 Le monitoring

La plateforme Space inclut un système de surveillance (*monitoring*) [57] permettant de recueillir diverses statistiques et métriques de performance sur les calculs et les communications pendant la simulation. Ce système de surveillance permet d'examiner (*to monitor*) les modules matériels et logiciels. Grâce à l'emploi de macro générique, la surveillance est non intrusive, elle ne change pas le temps de simulation et surcharge peu le temps physique. Les composants surveillés sont les modules utilisateurs matériels, les bus, les mémoires et les modules utilisateurs logiciels présents sur le simulateur de processeur (*ISS*). Les activités surveillées sont :

- transfert de module à module : l'information tracée comprend l'expéditeur et le destinataire du transfert, sa taille et sa durée et ceux aussi bien pour une écriture (*ModuleWrite()*) que pour une lecture (*ModuleRead()*).

- Transfert sur le bus : l'information tracée contient toujours la source et la destination du transfert, la taille, le type (écriture ou lecture) et la durée de transfert à travers le bus.

- Accès mémoire : l'information tracée comprend la source et la mémoire destinataire, le temps d'accès, l'adresse de l'emplacement mémoire accédé et la taille.

- Calcul dans les modules utilisateurs: l'information contient le temps de début et de fin d'un calcul quelconque (e.g. un algorithme de recherche, une multiplication). En logiciel, le nombre d'instructions consommées par le calcul est aussi enregistré.

Par la suite, toutes ces activités peuvent être observées et analysées après la simulation textuellement ou graphiquement. Ainsi, des informations telles que le nombre total de transactions, le nombre de requêtes d'écriture ou de lecture, le pourcentage d'utilisation du bus, le nombre d'accès mémoire par tel composant et bien d'autres renseignent le concepteur sur les caractéristiques de son application.

4.1.3 Bancs de test

4.1.3.1 Producteur-consommateur

Un banc de test de type producteur-consommateur (figure 4.1) a été utilisé avec un seul bus et deux bus reliés par un pont pour permettre la comparaison.

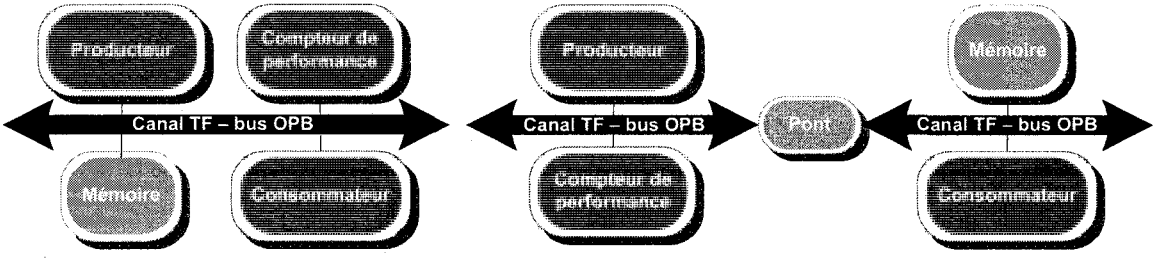

Figure 4.1 Architectures de communication pour la comparaison TF/BCA

Le Producteur exécute quatre séquences de tests. Chaque séquence comprend une

boucle avec une ou deux fonctions de communication. Le nombre d'itérations est configurable :

- 1ère séquence : le Producteur envoie une donnée de 4 octets (1 mot) au Consommateur. Le Producteur et le Consommateur utilise une communication bloquante à l'aide d'un *ModuleWrite()* associé à un *ModuleRead()*.
- 2ème séquence : le Producteur envoie une transaction de 17 mots (i.e. 17*4 octets) au Consommateur toujours à l'aide d'une communication bloquante.
- 3ème séquence : le Producteur écrit en mémoire une donnée de 4 octets (1 mot) par l'intermédiaire d'un *DeviceWrite()*.
- 4ème séquence : le Producteur envoie une donnée de 4 octets au Consommateur et écrit une donnée de 4 octets en mémoire.

Le Compteur de performance permet de calculer le temps physique et le temps de simulation et donne quelques statistiques sur la simulation. Le Producteur envoie des commandes au Compteur de performance pour que celui-ci prenne le temps avec la fonction *sc_simulation_time()* ou *clock()*.

Tableau 4.1 Configurations du banc de test Producteur-Consommateur

Nombre d'itérations	10 millions pour séquence 1, 2 et 3 5 millions pour séquence 4
Canal TF – bus OPB	32 bits, arbitrage FIFO, mode rafale, 25 MHz (soit 20 ns)
Pont fonctionnel et OPB-OPB	Mode direct, 1 cycle de latence de transfert

4.1.3.2 Application de décodage JPEG et de détections diverses

Afin d'utiliser la méthode d'exploration décrite à la partie 3.2 et d'étudier diverses architectures de communication, un modèle de décodeur JPEG est utilisé. Ce modèle est associé avec une détection de contour, faciale et des yeux. Les images à

décompresser sont donc des visages. Cette application multimédia de type traitement de données en continu (*streaming*) soumet l'architecture de communication à un trafic réel rencontré dans les systèmes-sur-puce. La figure 4.2 et le tableau 4.2 présentent les différents modules et leur rôle.

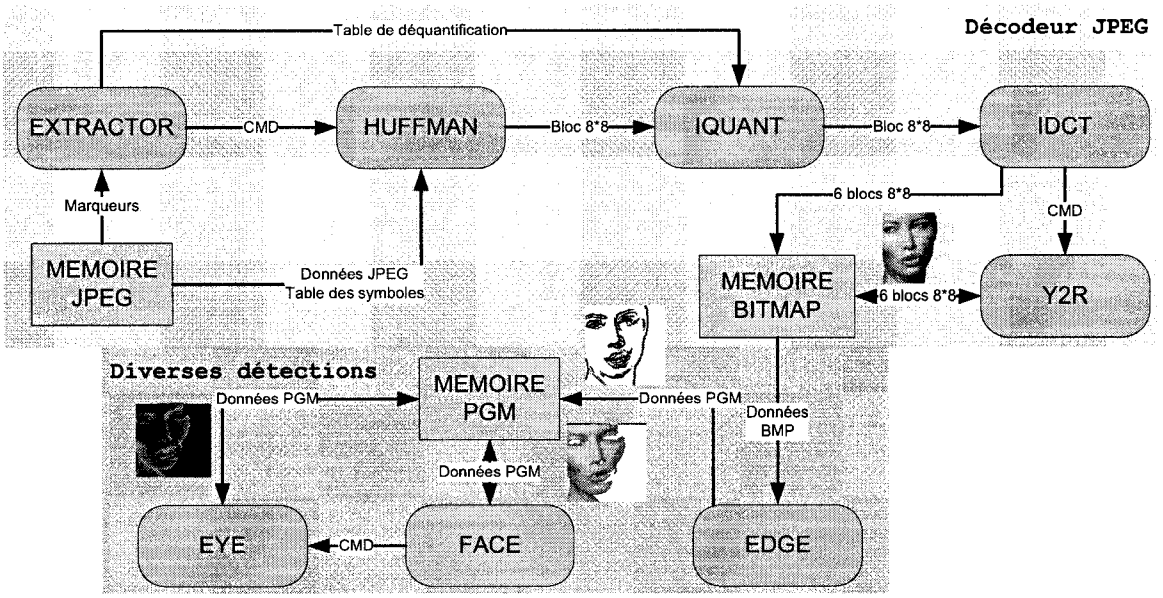

Figure 4.2 Application JPEG

Les modules s'échangent des commandes pour se synchroniser lors des différentes étapes. Les commandes sont basées sur une structure de 20 octets contenant 5 champs (type de la commande, 3 paramètres liés à la commande et le nombre de paramètre). La figure 4.2 montre deux chemins de données : le décodage JPEG et les diverses détections.

Communications des modules :

Afin de mieux comprendre les résultats de la partie 4.3, le tableau 4.3 présente toutes les communications effectuées par les différents modules (*ModuleWrite, ModuleRead, DeviceWrite* et *DeviceRead*) pour le traitement d'une image **JPEG** de **128*128 pixels** soit 16384 pixels.

Tableau 4.2 Modules JPEG et leur rôle

Module[1]	**Rôle**
Extractor (EX)	Lit l'entête de l'image *jpeg*, envoie certaines commandes à Huffman, la table de déquantification à Iquant, lance les détections (Edge et Face) et arrête la simulation après un certain nombre d'images.
Huffman (HU)	Lit la table de symboles et les données *jpeg*, effectue la décompression de Huffman à partir de la table de symboles et envoie les blocs de 8*8 éléments à Iquant.
Iquant (IQ)	Réceptionne les blocs 8*8, effectue la quantitification inverse à partir de la table et envoie ces mêmes blocs à IDCT.
IDCT (ID)	Réceptionne les blocs 8*8, effectue une transformée inverse discrête en cosinus sur 6 blocs, les écrit en mémoire et prévient Y2R.
Y2R (Y2)	Permet le changement d'espace de couleurs de YUV à RGB. Lit les 6 blocs 8*8 pixels en mémoire (4 blocs de luminance et 2 blocs de chrominance), effectue le changement d'espace et écrit l'image *bmp* en mémoire.
Edge (ED)	Lit les données *bmp*, effectue la détection de contour, écrit l'image contour au format *pgm* en mémoire.
Face (FA)	Lit les données *pgm* de l'image contour, effectue la détection faciale, écrit l'image faciale au format *pgm* en mémoire et lance Eye.
Eye (EY)	Lit les données *pgm* de l'image faciale, effectue la détection des yeux, écrit l'image au format *pgm* en mémoire.

[1] Entre parenthèse apparaît les noms des modules utilisés dans les différentes configurations de tests. Les mémoires sont nommées JPG, BMP et PGM.

Tableau 4.3 Communications des modules JPEG

Module	ModuleWrite	ModuleRead	DeviceWrite	DeviceRead
EXTRACTOR	150	105	2	199
HUFFMAN	6436	100	0	1154
IQUANT	6435	6489	2	2
IDCT	16	6435	3072	0
Y2R	0	16	4096	3074
EDGE	1	1	4098	4098
FACE	1	1	4096	4096
EYE	1	1	4096	4096

À partir des tableaux 4.2, 4.3 et de la figure 4.2, nous pouvons retrouver la dynamique de l'application et déduire trois groupes de modules :

- communications module-module majoritaires : HUFFMAN et IQUANT.
- communications module-mémoire majoritaires : Y2R, EDGE, FACE et EYE.
- communications module-module/mémoire assez équilibré : EXTRACTOR et IDCT.

4.2 Comparaison des estimations de simulation au niveau TF et BCA

Le but est d'étudier la performance de simulation obtenue au niveau TF et au niveau OPB. L'estimation des latences de communication dans le canal TF et le pont fonctionnel nous apporte-t-il des résultats satisfaisants par rapport à la simulation avec un bus OPB et un pont OPB-OPB au niveau BCA ? Les bancs de tests Producteur-Consommateur et application JPEG (2 images 128*128) ont été utilisés (cf. partie 4.1.3).

4.2.1 Analyse des résultats

Le tableau 4.4 présente le facteur d'accélération de la simulation ainsi que le pourcentage de justesse de l'estimation des communications au niveau TF par rapport au niveau BCA. Ces résultats ont été calculés à partir des tableaux VI.2, VI.3 et VI.4 en annexe VI. Les séquences 1 à 4 du banc de test Producteur-Consommateur correspondent à un trafic idéal avec un seul module qui accède au bus et l'application JPEG apporte un trafic réel dans lequel plusieurs modules peuvent accéder au bus.

Accélération :

Tableau 4.4 Comparaison des performances de simulation entre le niveau TF et BCA

Application	**Séq. 1**[1]		**Séq. 2**[1]		**Séq. 3**[1]		**Séq. 4**[1]		**JPEG**	
Nbre de bus	1	2	1	2	1	2	1	2	1	2^2
Accélération	2,16	2,17	2,65	4,33	1,88	2,03	2,04	2,04	2,38	2,89
Estimation	100	100	100	100	99,39	99,69	99,78	99,90	82,73	90,07

[1] banc de test Producteur-Consommateur

[2] Extractor, Huffman, Iquant, IDCT, Y2R, mémoire JPEG sur TF1/OPB1; Edge, Face, Eye, mémoire PGM sur TF2/OPB2; mémoire BITMAP sur TF1-TF2 et OPB1-OPB2

Dans toutes les applications, le réseau de communication au niveau TF permet une simulation plus rapide que ce même réseau au niveau BCA. On remarque que le gain varie en fonction du type de communication et du trafic.

Dans sa globalité, le gain obtenu au niveau TF s'explique par les mécanismes SystemC utilisés, i.e abstraction des détails des transactions et notion de temps. Le modèle de communication TF englobe toutes les phases d'une transaction (arbitrage, transfert acquittement) simulées par un seul et unique *wait()* avec un temps alors que le modèle OPB considère chacune de ces phases avec un *wait()* sur un front d'horloge. Le simulateur SystemC est un simulateur évènementiel. Plus le simulateur doit gérer des événements, plus la simulation sera longue. Les *wait* avec un temps ne font intervenir qu'un seul évènement. Les *wait* sur des fronts d'horloge font appel au canal hiérarchique **sc_clock** qui modélise le comportement d'un signal d'horloge. Ce comportement est réalisé grâce au canal primaire **sc_signal<bool>** et à deux processus SC_METHOD, un pour chaque front de l'horloge. Chaque front de l'horloge entraîne la notification d'un évènement pour l'autre front. Il y a donc 2 évènements à gérer pour un cycle de simulation. Cette gestion d'évènements implique une activité accrue du simulateur pour ordonnancer tous ces processus. On constate donc que l'utilisation d'une horloge *sc_clock* et du *wait* sur des fronts d'horloge peut ralentir la simulation SystemC. Plutôt que de coder un *wait* sur **N** fronts montants d'horloge. i.e **N wait(clk->posedge_event())** (niveau BCA), il est plus efficace de simuler une latence par un wait de **N*periode_horloge** (niveau

TF) avec periode_horloge de type *sc_time*.

L'explication précédente associée au découpage phase par phase d'un transfert (cf. partie 3.3.1.1) sur le bus OPB justifie le fait que la meilleure accélération est obtenue pour la séquence 2. En effet, dans la séquence 2, une transaction de 18 mots (1 mot d'entête Space + plusieurs données) est simulée par un seul *wait* dans le canal TF ainsi que dans le pont fonctionnel. Chaque wait entraîne la notification d'un évènement. Alors que dans le bus OPB et le pont OPB-OPB, la transaction de 18 mots utilise plusieurs **wait(clk->posedge_event())**. Dans les séquences 1 et 3, l'accélération est moins élevée car la transaction n'étant que de 2 mots (écriture à un module) et de 1 mot (écriture à une mémoire) dans le canal TF, un seul évènement est généré par *wait* mais ce *wait* est plus court. Sur le bus OPB, la transaction se limite donc à très peu de **wait(clk->posedge_event())** par phase du transfert. La séquence 4 combine les effets des modélisations du temps par des *wait* de la séquence 1 et 3.

Dans l'application JPEG, il y a un mélange des divers types de communication offerts par Space. Les transactions sur le bus varient entre 1 et 6 mots en taille. Au niveau TF, les communications vers des modules ou des mémoires avec des tailles de transactions importantes accélèrent la simulation alors que les transactions de faible taille diminuent cette accélération. Un trafic réel combine les avantages et les inconvénients des divers types de communication. Les explications pour les séquences 1 à 4 sont regroupées ici.

Estimation des communications :

Le canal TF permet une bonne estimation des communications. Les résultats valident l'utilisation des formules dans la partie 3.3.1.2. La formule liée au mode rafale reproduit da façon adéquate le mode pseudo-rafale (signal *sequential + bus lock*) du bus OPB. La formule prend en compte que seul un arbitrage est nécessaire pour l'ensemble des transferts constituant la transaction. Un entête est ajouté à toutes les

transactions dans le bus OPB pour une communication entre modules. Le paramètre, accès supplémentaire, pour le canal TF permet de tenir compte de l'entête. Ceci permet d'obtenir des estimations de 100% et 99% pour les séquences 1 à 4. Cependant, lorsque le modèle TF est soumis à un trafic réel de données, il permet d'obtenir une estimation supérieure à 80%. La raison est que le canal TF ne modélise pas les délais d'attente des modules voulant accéder au bus alors que celui-ci est pris par un autre module. En effet, dans le modèle OPB, lorsqu'un module veut le bus et qu'un transfert est déjà en cours, sa requête est mise dans une liste d'attente. L'arbitre se réveille alors pour arbitrer cette requête mais comme le bus est utilisé, il attend un front montant d'horloge. Ainsi de suite jusqu'à ce que le bus soit libéré. Tandis que dans le modèle TF, l'arbitre va se réveiller mais si le bus est occupé, il attend un événement indiquant que le bus est libre.

L'histogramme de la figure 4.3 (issu des tableaux VI.2 et VI.3) montre les performances du canal TF et du bus OPB pour les séquences 1 à 3 du banc de test Producteur-Consommateur dans la configuration simple bus et multibus. Les colonnes pour le canal TF soulignent que celui-ci achève la meilleure performance de simulation lorsque les transactions sont de taille importante ($>$ à 1 mot). Ici, il est illustré pour des *ModuleWrite()* mais ceci est vrai aussi pour les *DeviceWrite()* et *DeviceRead()*. L'explication réside dans l'unique *wait* présent dans le canal pour simuler toute la transaction.

Les colonnes pour le canal OPB montrent que celui-ci est le plus performant pour les communications vers les périphériques (*DeviceWrite()* et *DeviceRead()*). En effet, ces dernières ne requièrent pas d'entête au message ni d'acquittement Space (présent pour les écritures bloquantes de module). Ce qui limite les *wait* sur des fronts d'horloge et accélère la simulation.

Conclusion :

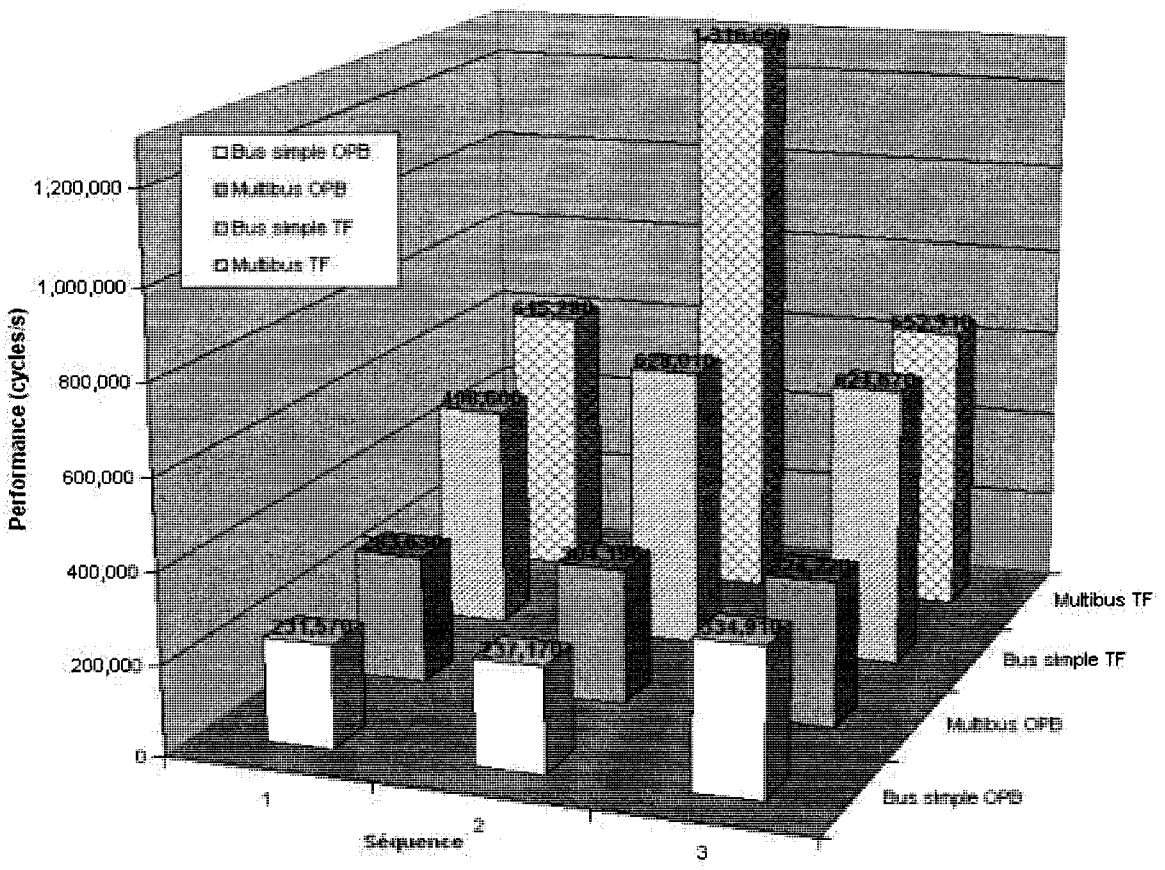

Figure 4.3 Performance de l'architecture de communication au niveau TF et BCA

La performance de simulation d'un modèle SystemC dépend du style de modélisation. Une abstraction des détails de communication et l'utilisation adéquate de la notion de temps peuvent améliorer cette performance de simulation. Lors d'un trafic idéal (application Producteur-Consommateur), le modèle TF apporte une accélération importante pour des transactions supérieures à un mot avec une estimation de 100% car aucun phénomène dynamique n'est présent dans le modèle OPB. En revanche, lorsque le modèle de communication TF est soumis à un vrai trafic, l'estimation des communications est moindre à cause de certains phénomènes dynamiques non reproduits (i.e. attente des modules pour accéder au bus) mais reste supérieure à 80%. L'accélération apportée par le modèle TF dépend de la taille de la transaction. Plus la transaction est importante, plus l'accélération augmente. Même si les mécanismes

de SystemC permettent d'accélérer le modèle TF en utilisant certaines techniques de modélisation SystemC, le niveau de détails du modèle OPB reste assez abstrait pour permettre des simulations rapides. Le modèle OPB ne modélise pas le comportement du bus de façon précise au signal mais utilise un niveau de détails des communications par regroupement de signaux. Aussi, le fait de remplacer le canal TF par un modèle OPB n'augmente pas dramatiquement le nombre de processus à gérer par l'ordonnanceur SystemC (1 processus pour le canal TF, 2 processus pour le bus OPB). Les changements de contexte des processus peuvent parfois ralentir la simulation si ces derniers sont nombreux, ce que démontre l'ajout d'un sc_clock. Un moyen d'accélérer la simulation au niveau TF serait de bouger de grande quantité de données. Dans l'application JPEG, au lieu d'écrire en mémoire pixel par pixel, les blocs de 64 pixels pourraient directement être écrits en une seule fois dans la mémoire. Ainsi un seul *wait* simulerait le transfert du bloc au lieu de un *wait* par pixel.

4.3 Performance à travers la méthodologie d'exploration

À partir de l'exemple du décodage JPEG (cf. partie 4.1.3.2), l'objectif est d'améliorer la performance du système en trouvant la meilleure architecture de communication possible. Dans le cas où un certain partitionnement est utilisé pour évaluer la performance de l'architecture de communication, le partitionnement est fait de façon ad hoc (i.e. selon l'expérience du concepteur). Ainsi, pour la phase 3 de la méthode d'exploration, la configuration pourra être toute matérielle ou partitionnée (matériel et logiciel).

4.3.1 Performance selon le type de topologie

Dans toutes les configurations de l'application JPEG, le système fonctionne avec une **horloge de période 20 ns**. Les tests portent sur le traitement de **2 images 128*128 pixels** pour permettre un pipeline des deux chemins de données. Les ponts sont bidirectionnels entre chaque paire de bus.

4.3.1.1 Réseau multibus (phase 2 de la méthodologie)

Configuration :

En appliquant la phase 2 de la méthode d'exploration (cf. partie 3.2), plusieurs architectures de communication basées sur des bus sont étudiées. Le but est de trouver une architecture offrant des performances satisfaisantes par rapport à une configuration initiale et d'utiliser le niveau TF pour exécuter des simulations rapides. Les différentes configurations dans lesquelles les modules de l'application JPEG sont répartis sont montrées dans le tableau 4.5.

Tableau 4.5 Configurations monobus et multibus

Configuration	**Bus 1**	**Bus 2**	**Bus 3**
Monobus	EX HU IQ ID Y2 JPG ED FA EY BMP PGM	-	-
Multibus 1 (aléatoire)	EX IQ Y2 JPG FA	HU ID ED EY BMP PGM	-
Multibus 2 (*clustering*)	EX HU IQ ID Y2 JPG	ED FA EY BMP PGM	-
Multibus 3 (*clustering*)	EX HU IQ ID Y2 JPG BMP	ED FA EY PGM	-
Multibus 4 (*clustering*)	EX HU IQ JPG	ED FA EY BMP PGM ID Y2	-
Multibus 5 (*clustering*)	EX HU IQ JPG	ED FA EY PGM	ID Y2 BMP

La configuration '**Monobus**' est l'architecture de base avec un bus partagé sim-

ple. Cette configuration est la **référence** pour améliorer les performances. La configuration 'Multibus 1' est une architecture multibus où les modules ont été placés aléatoirement. Par la suite, les autres configurations sont des architectures multibus dans lesquels les modules ont été distribués en tenant compte des concepts de regroupement des communications (*clustering*) et de la notion de concurrence.
Le canal TF et le pont fonctionnel sont paramétrés comme ceci :

- Canal TF : taille de 32 bits, fréquence de 50 MHz, arbitrage FIFO, latences du bus OPB (2 cycles d'arbitrage, 1 cycle de décodage d'adresse, 1 cycle de transfert/acquittement), 1 accès supplémentaire pour l'entête Space.
- Pont fonctionnel : mode Direct, latence de transfert de 1 cycle, latence supplémentaire de 1 cycle pour le l'entête Space.

Résultats :

L'histogramme de la figure 4.4 présente le nombre de cycles simulés pour chaque configuration du tableau 4.5. Les résultats montrent que les configurations 'Multibus 2' et 'Multibus 3' offrent la meilleure performance avec 2 347 196 et 2 557 900 cycles respectivement. Elles apportent une amélioration de **14,6** et 6,9% par rapport à 'Monobus'. La configuration 'Multibus 1' détériore la performance de 54,7%. Tandis que 'Mutlibus 4' et 'Multibus 5' ne détériore la performance de l'architecture à bus partagé simple que de 1,2% et 0,6% respectivement.

Les résultats de l'histogramme de la figure 4.4 s'explique par le fait que dans un système multibus, les ponts allongent le temps de la transaction qui doit traverser deux bus. Il faut donc limiter les transactions à travers le pont, i.e. communications **inter-bus**, et favoriser les communications **intra-bus**. Dans le cas 'Multibus 2', la mémoire BITMAP est sur le Bus 2. Sur le Bus 1, IDCT et Y2R doivent passer par le pont pour écrire et lire cette mémoire mais les transactions du Bus 2 vers le Bus 1 sont négligeables. Un seul sens est privilégié à travers le pont ce qui donne une bonne

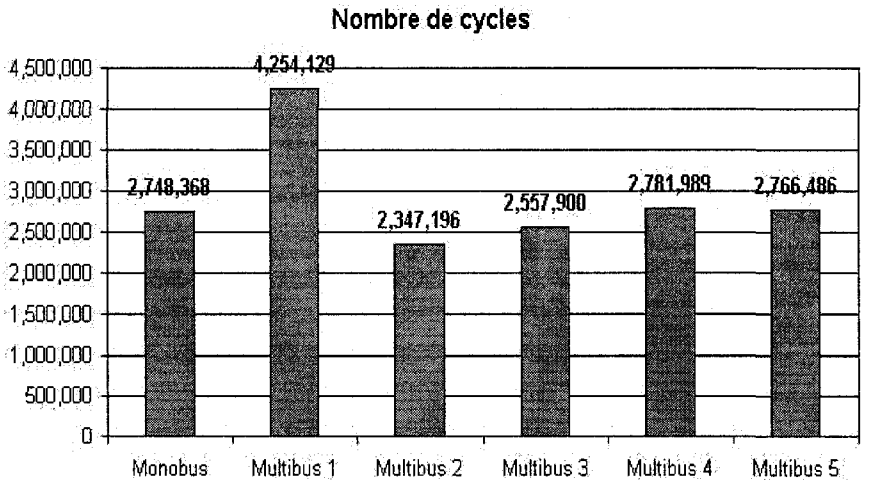

Figure 4.4 Nombre de cycles simulés pour différentes configurations multibus TF

performance. Le même raisonnement s'applique pour 'Multibus 3' avec la mémoire BITMAP sur le Bus 1 et Edge qui accède à cette mémoire par le pont entre le Bus 2 et 1. Pour les configurations 'Multibus 1' et 'Multibus 4', les communications engendrent des communications interbus bidirectionnelles importantes, ce qui augmente la latence de beaucoup de communication et dégrade la bande passante des deux bus en même temps. Dans le cas 'Multibus 5', la répartition des modules sur 3 bus n'améliore pas la performance car l'application JPEG ne présente pas un parallélisme fort entre les modules. Même en regroupant certains modules communiquant beaucoup entre eux, il y a toujours une certaine exécution séquentielle des étapes et l'ajout de ponts augmente les latences de communication.

Les regroupements de base des modules sur un bus suivent le raisonnement suivant : les deux chemins de données (JPEG et détections) permettent un regroupement évident des modules pour le *clustering*. Les modules JPEG sur un bus avec la mémoire JPEG et les modules de détection sur un autre avec la mémoire PGM. La mémoire BITMAP étant utilisée pour la transition entre la décompression et la détection, cela pose problème. Elle peut être sur un bus ou un autre. L'application introduit du parallélisme avec une méthode de pipeline. Le pipeline se situe au niveau de plusieurs

images décompressées, i.e. dès que les détections de contour commencent pour la 1ère image décompressée, les modules JPEG peuvent traiter une 2ème image. Cela conduit encore à la répartition des modules introduite par le *clustering*.

L'histogramme de la figure 4.5 présente l'utilisation des bus pour chaque configuration du tableau 4.5. Nous voyons que toutes les configurations multibus désengorgent le Bus 1 utilisé à 60% dans l'architecture simple bus. Dans 'Multibus 1' et 'Multibus 4', l'utilisation du Bus 1 et du Bus 2 s'approche d'un équilibre, environ 35% mais cela ne permet d'améliorer la performance. 'Multibus 2' et 'Multibus 3' permettent une utilisation assez forte du Bus 1 (>50%) avec une utilisation moyenne du Bus 2 qui mène au meilleur résultat. Le 'Multibus 5' partagent les communications entre les 3 bus mais ne permet pas forcément une bonne utilisation de tous les bus.

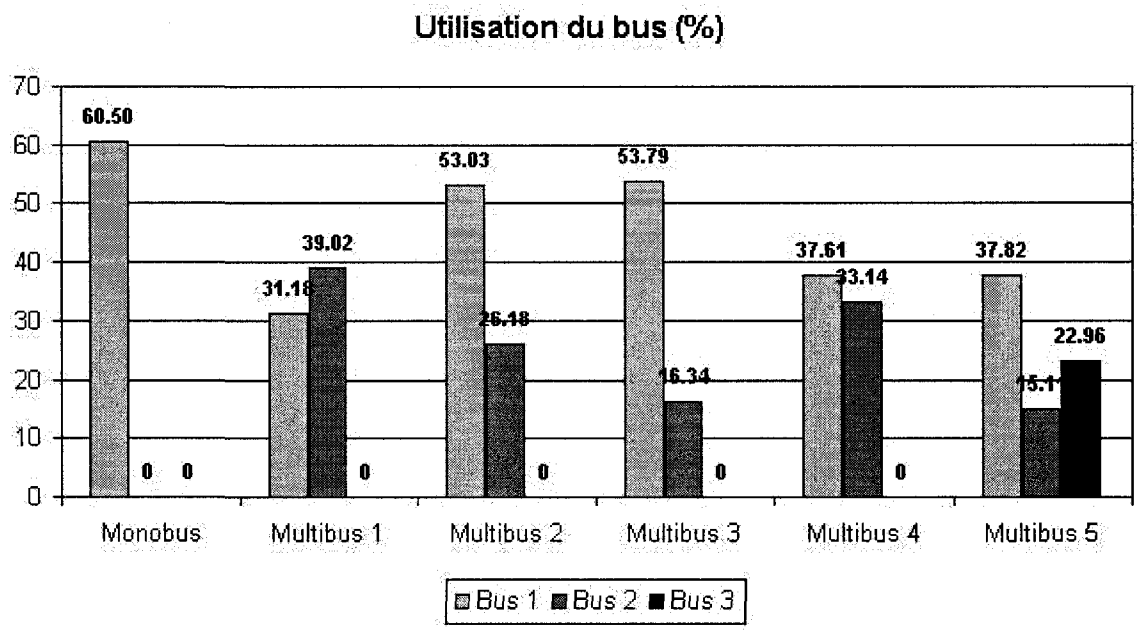

Figure 4.5 Utilisation du canal TF pour différentes configurations multibus

L'utilisation du bus dans toutes les configurations multibus dépend des communications inter- et intra-bus engendrés par les modules.

Dans les configurations 'Multibus 1' et 'Multibus 4', les communications inter-bus

importantes à travers les ponts dans les deux sens favorisent un certain équilibre dans l'utilisation des bus. Dans le cas 'Multibus 1', le Bus 2 est utilisé à environ 40% car la présence de Huffman, IDCT et de la mémoire BITMAP favorise les transactions à travers le pont et IDCT, Edge et Eye communiquent beaucoup vers les mémoires de ce bus comparativement aux autres modules sur le Bus 1. Entre les configurations 'Multibus 2' et 'Multibus 3', la mémoire BITMAP a été déplacée du Bus 2 vers le Bus 1. Ceci a pour conséquence de diminuer l'utilisation du Bus 2 pour le 'Multibus 3' car les communications sont plus abondantes sur le Bus 1. Pour passer du cas 'Multibus 4' au cas 'Multibus 5', les modules du Bus 2 ont été répartis sur deux bus (Edge, Face, Eye et mémoire PGM sont restés sur le Bus 2 alors que IDCT, Y2R et mémoire BITMAP sont sur le Bus 3). Nous voyons que l'utilisation du Bus 1 est quasiment la même dans les deux cas. Le Bus 3 est plus utilisé que le Bus 2 car IDCT, Y2R et la mémoire BITMAP engendrent plus de communications à travers les ponts entre le Bus 1 et 3 (IDCT reçoit des blocs 8*8 de Iquant et renvoit des acquittements Space) et entre le Bus 2 et 3 (Edge lit la mémoire BTIMAP).

Conclusion :

La méthode de regroupement des communications (*clustering*) associé à la concurrence des modules apporte de bonne performance pour palier à la contention sur un bus partagé simple. Une application de type pipeline est parfois limitée par les communications à travers les ponts qui allongent le temps de transferts des transactions. L'utilisation adéquate des bus dépend du placement des modules et des communications qu'ils engendrent. Des communications inter-bus dans les deux sens à travers les ponts peuvent équilibrer les bus sans toutefois apporter une bonne performance. Le graphe démontre qu'une distribution aléatoire des modules ne mène pas à une optimisation de la performance (Multibus 1) et qu'une distribution par *clustering* permet de meilleurs résultats (Multibus 2, 3, 4 et 5).

4.3.1.2 Composants annexes (phase 3 de la méthodologie)

La phase 2 (niveau TF) a permis de sélectionner la configuration 'Multibus 2' comme meilleure architecture de communication (cf. table 4.5 pour le placement des modules). Une simulation fonctionnelle de cette architecture au niveau BCA est effectuée et les résultats de performance deviennent la nouvelle référence à améliorer. Ici, nous nous intéressons à l'amélioration de la topologie multibus par l'utilisation des composants annexes (liens point à point et mémoire double port). L'amélioration de la topologie par la configuration du protocole est expliquée à la partie 4.3.2. La phase 3 est choisie pour cette exploration car elle permet d'obtenir des résultats plus proches de la réalité. Tout est connu cycle par cycle sur le bus.

Configuration :

Toutes les configurations sont matérielles. Les différentes configurations dans lesquelles sont associés le multibus et/ou les liens point à point et/ou la mémoire double port sont montrées dans le tableau 4.6. Pour toutes les configurations, les modules de l'application JPEG sont répartis comme suit : **Extractor, Huffman, Iquant, IDCT, Y2R et la mémoire JPEG sur le bus OPB1 et Edge, Face, Eye et la mémoire PGM sur OPB2**. La mémoire BITMAP subit un traitement particulier. La configuration 'Multibus 2' est la **nouvelle référence** pour améliorer les performances. Toutes les autres configurations sont des variantes de 'Multibus 2'.

Le bus OPB, le pont OPB-OPB et le lien point à point ont les caractéristiques suivantes :

- Bus OPB: taille de 32 bits, fréquence de 50 MHz, arbitrage PRIORITÉ, latences (2 cycles d'arbitrage, 1 cycle de décodage d'adresse, 1 cycle de transfert/acquittement).
- Pont OPB-OPB : latence de transfert de 1 cycle.

Tableau 4.6 Configurations des réseaux multibus avec les composants annexes

Configuration	**Caractéristiques spécifiques**
Multibus 2	BMP sur OPB2
Multibus 2a	BMP sur OPB1 ET OPB2 (mémoire double port)
Multibus 2b	BMP sur OPB1 ET OPB2 (mémoire double port) IQ point à point avec IDCT[1]
Multibus 2c	BMP sur OPB1 ET OPB2 (mémoire double port) HU point à point avec IQ IQ point à point avec IDCT
Multibus 2d	BMP sur OPB1 ET OPB2 (mémoire double port) EX point à point avec HU et IQ HU point à point avec IQ IQ point à point avec IDCT

[1] Signifie : Iquant est relié à IDCT par un lien point à point bidirectionnel

- Lien point à point : 1 cycle de latence pour écriture et lecture.

Résultats :

Le graphe 4.6 présente le nombre de cycles simulés pour chaque configuration du tableau 4.6. Les résultats montrent que la configuration 'Multibus 2a' (mémoire double port) apporte une amélioration de la performance de **20,5%** par rapport à 'Multibus 2'. Les configurations 'Multibus 2b', 'Multibus 2c' et 'Multibus 2d' (liens point à point + mémoire double port) améliore la performance respectivement de 35,4%, 57,1% et **57,3%**.

En connectant la mémoire BITMAP sur OPB1 et OPB2 (Multibus 2a), le parallélisme de l'application JPEG est grandement amélioré. En effet, en laissant la mémoire BITMAP sur OPB2, IDCT et Y2R doivent passer par le pont OPB-OPB engendrant beaucoup de communications inter-bus qui se heurtent aux communications intra-bus de Edge, Face et Eye dans les mémoires. Le pont devient alors un goulot d'étranglement. Les configurations 'Multibus 2b', 'Multibus 2c' et 'Multibus 2d' montrent de façon évidente que des liens directs entre modules accélèrent les communications. De nombreux cycles d'arbitrage sur le bus OPB1 sont économisés.

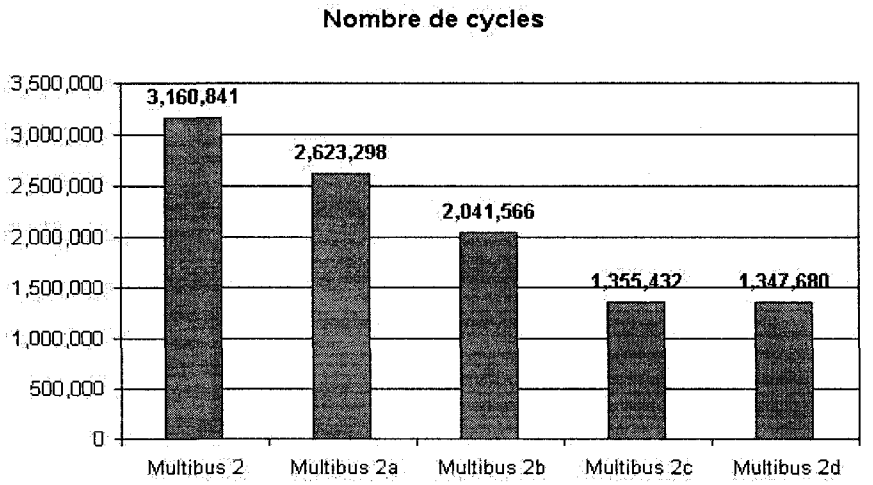

Figure 4.6 Nombre de cycles simulés pour différentes configurations multibus BCA

Les liens point à point ont été établis entre les modules communiquant ensemble. Il s'agit de Extractor, Huffman, Iquant, et IDCT. Les liens point à point sont intéressants si le volume de données échangées entre modules est important, ce que montre 'Multibus 2b' et 'Multibus 2c' (les couples Iquant-IDCT et Huffman-Iquant échangent beaucoup de blocs 8*8). En comparant 'Multibus 2c' et 'Multibus 2d', on voit que les liens directs de Extractor apportent peu d'amélioration. Les communications de ce module pourraient passer par le bus OPB. Dans les détections de contour, les modules font uniquement des accès mémoires. Ils ne sont donc pas concernés par les liens point à point.

L'histogramme de la figure 4.7 présente l'utilisation des bus pour chaque configuration du tableau 4.6. L'utilisation de la mémoire BITMAP double port dans 'Multibus 2a' diminue les communications inter-bus à travers le pont OPB-OPB. D'où l'utilisation moindre de OPB2 'Multibus 2a' par rapport à 'Multibus 2'. Les communications à travers un pont allonge le nombre de cycles pendant lesquels le premier bus est occupé, ce qui entraine une forte utilisation de celui-ci (OPB1 à 59,5% dans 'Multibus 2'). La tendance observée pour les liens point à point est la diminution de l'utilisation du bus OPB1 (Multibus 2b, 2c et 2d). En effet, moins de communication ont lieu sur le bus

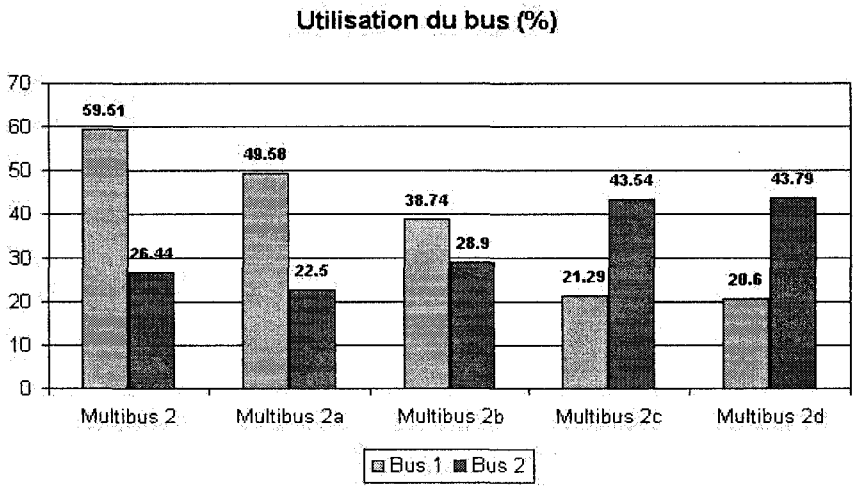

Figure 4.7 Utilisation du bus OPB pour différentes configurations multibus

au profit de communication directe entre les modules. Le pourcentage d'utilisation du bus OPB2, quand à lui, augmente mais cela ne veut pas dire que OPB 2 est plus utilisé. Cela est dû au temps d'exécution qui raccourcit mais comme la proportion des communications sur OPB2 reste la même, le rapport entre le temps totale des communications sur OPB2 et le temps d'exécution augmente.

Conclusion :

L'utilisation de mémoire double port améliore le parallélisme d'un réseau multibus. À condition que l'application permette son utilisation. Une mémoire double port aide les modules à s'échanger de grande quantité de données en utilisant quelques messages de synchronisation pour ne pas que deux modules arrivent en même temps sur la même zone mémoire pour écrire ou écrire/lire. Les liens point à point améliorent efficacement la performance du système en désengorgeant le bus. Il faut favoriser les liens directs entre modules s'échangeant un volume de données important.

4.3.1.3 Liens point à point et processeur (phase 3 de la méthodologie)

Le but est de voir comment les liens point à point permettent d'améliorer l'architecture de communication en utilisant différents partitionnements matériel/logiciel des divers modules.

Configuration :

Les différentes configurations dans lesquelles sont associés le multibus et/ou les liens point à point dans le tableau 4.7. Pour toutes les configurations à deux bus OPB, les modules de l'application JPEG sont répartis comme suit : **Extractor, Huffman, Iquant, IDCT, Y2R et la mémoire JPEG sur le bus OPB1; Edge, Face, Eye et la mémoire PGM sur OPB2; la mémoire BITMAP sur OPB1 et OPB2**. Pour la configuration à trois bus, le placement est : **Extractor, Huffman, Iquant, mémoire JPEG sur OPB1; IDCT, Y2R sur OPB3; Edge, Face, Eye et la mémoire PGM sur OPB2; la mémoire BITMAP sur OPB2 et OPB3**. Les caractéristiques du bus OPB et du pont OPB-OPB sont les mêmes que dans la partie 4.3.1.2. Les liens point à point ont la latence suivante :

- Processeur-module et processeur-processeur : 2 cycles
- Module-module : 1 cycle

Résultats :

Le tableau 4.8 présente le nombre de cycles simulés pour chaque configuration du tableau 4.7. Afin d'étudier le gain apporté par l'utilisation des liens point à point avec un réseau multibus monoprocesseur ou multiprocesseur, des simulations multibus sans liens point à point ont été effectués (Multibus 2e1, 2f1, 2g1 et 2h1). Les autres configurations contiennent des liens point à point.

Les résultats démontrent que l'utilisation de liens point à point entre un processeur

Tableau 4.7 Configurations des partitionnements matériel/logiciel

Configuration	**Caractéristiques spécifiques**
Multibus 2e1	EX en logiciel
Multibus 2e2	EX en logiciel EX point à point avec HU et IQ[1]
Multibus 2e3	EX en logiciel Tout point à point[2]
Multibus 2f1	IDCT en logiciel
Multibus 2f2	IDCT en logiciel IQ point à point avec IDCT
Multibus 2f3	IDCT en logiciel Tout point à point[2]
Multibus 2g1	IDCT et EX en logiciel sur 2 processeurs différents
Multibus 2g2	IDCT et EX en logiciel sur 2 processeurs différents Tout point à point[2]
Multibus 2h1	IDCT et IQ en logiciel sur le même processeur
Multibus 2h2	IDCT et IQ en logiciel sur le même processeur Tout point à point[2]

[1] « IQ point à point avec IDCT » veut dire Iquant est relié à IDCT par un lien point à point

[2] EX point à point avec HU et IQ, HU point à point avec IQ, IQ point à point avec IDCT

Tableau 4.8 Performance des différents partitionnements matériel/logiciel

Configuration	**Nombre de cycles**	**Accélération**	**Temps physique (s)[1]**
Multibus 2e1	33 771 036	1	191
Multibus 2e2	12 427 539	2,71	89
Multibus 2e3	12 257 469	2,75	76
Multibus 2f1	938 974 610	1	3 141
Multibus 2f2	164 471 133	5,71	663
Multibus 2f3	164 466 294	5,71	649
Multibus 2g1	942 677 431	1	6 010
Multibus 2g2	166 641 432	5,65	1 269
Multibus 2h1	2 692 575 926	1	9208
Multibus 2h2	1 546 471 653	1,74	6979

[1] ce temps comprend le *monitoring* textuel

et un module matériel améliore la performance de l'application JPEG. Par exemple, pour les systèmes monoprocesseurs, la performance en terme de cycles est améliorée de 63,7% ('Multibus 2e3' par rapport à 'Multibus 2e1' - Extractor en logiciel) et de 82,4% ('Multibus 2f3' par rapport à 'Multibus 2f1' - IDCT en logiciel). Dans un système multiprocesseur, les liens point à point entre processeur et module matériel améliorent de 82,3% la performance ('Multibus 2g2' par rapport à 'Multibus 2g1' - Extractor et IDCT en logiciel) et de 42,5% ('Multibus 2h2' par rapport à 'Multibus 2h1' - Iquant et IDCT en logiciel sur le même processeur).

Ceci s'explique par les mécanismes de communication mis en jeu par un bus et par un lien point à point. Il y a deux types de communication à considérer : modules matériel->logiciel (HW->SW), logiciel->matériel (SW->HW)[2]. Reportez-vous à la partie 3.1.1 pour plus de détails sur les communications Space et à la figure de l'annexe III pour avoir une représentation graphique des composants.

- Par le bus OPB :
  - Pour chaque type de communication, que ce soit le processeur ou le module matériel qui initie la transaction, ces derniers doivent accéder au bus OPB. Avant d'envoyer leur transaction, ils peuvent attendre un certain délai si le bus est occupé et avant que l'arbitre les sélectionne. Ce délai d'attente peut prendre de nombreux cycles si plusieurs modules se partagent le bus OPB.
  - Pour les écritures bloquantes, le processeur ou le module matériel doivent attendre l'acquittement Space qui est une transaction à part entière sur le bus. Il faut donc compter toute la latence d'un transfert sur le bus pour que le processeur ou le module matériel soient débloqués.

[2]Le lien point à point entre 2 processeurs n'a pas pu être testé car il n'était pas disponible au moment de l'expérimentation.

  - Le processeur effectue de nombreux accès en lecture et écriture dans le PIC et le gestionnaire de communication. Dans le PIC, les accès servent à connaître le module à l'origine d'une interruption et à acquitter une interruption traitée. Dans le gestionnaire de communication, les accès aident à récupérer les données pour le module logiciel. Tous ces accès passant par le bus OPB entraînent de nombreux cycles supplémentaires pour traiter une communication Space dans sa globalité et ralentissent le traitement d'une communication par le processeur.

- Par un lien point à point (i.e. FSL):

  - Pour chaque type de communication, il n'y a pas de délai d'attente ni d'arbitrage. Le lien est direct et non partagé. Beaucoup de cycles sont donc épargnés.
  - Pour les écritures bloquantes, il n'y a pas d'envoi d'acquittement Space sur un bus. Le module logiciel sur le processeur est débloqué par une interruption du lien FSL. Le module matériel est débloqué par évènement SystemC. Ce traitement est donc plus rapide et économise encore des cycles d'horloge.
  - Le processeur effectue des accès dans le PIC pour traiter le caractère bloquant des liens point à point. Les accès dans le PIC sont beaucoup moins nombreux que pour une communication par le bus. Il n'y a plus d'accès dans le gestionnaire de communication. Le lien point à point accélère le traitement d'une communication.

Un lien point à point offre donc une gestion plus rapide des communications entre un module matériel et logiciel sur un processeur.

Lorsque Extractor est en logiciel, les liens point à point apportent une accélération autour de 2,7 (Multibus 2e1 et 2e3). De la même façon, l'accélération tourne autour

de 5,7 avec IDCT en logiciel (Multibus 2f1 et 2f3). IDCT est le plus rapide car le volume de données reçues de Iquant par le lien point à point est important (cf. tableau 4.3 et figure 4.2). Il ne fait que des lectures bloquantes sur Iquant dont l'exécution reste rapide. Les interruptions générées par le lien point à point ne ralentissent pas tellement l'exécution du module IDCT sur le processeur. Par la suite, le traitement des données et leur écriture en mémoire BITMAP par le processeur sont aussi rapides. Les accès mémoire par le processeur prennent le temps le plus court parmi toutes les communications possibles pour un module logiciel.
Le module Extractor fait une majorité d'écritures non bloquantes et de lectures bloquantes ainsi que quelques écritures bloquantes avec Huffman et Iquant en utilisant les liens point à point. Ces communications se limitent à l'envoi de message de synchronisation de 20 octets et de la table de déquantification. Par conséquent, il y a peu d'interruption qui interrompt le processeur pour Extractor, ce qui permet une exécution rapide de ce module logiciel mais le volume de données échangées par le lien point à point n'est pas assez important pour accélérer drastiquement l'exécution de Extractor d'où l'accélération autour de 2,7.
Ceci indique que les liens point à point sont très adaptés si les modules s'échangent beaucoup de données. Les liens point à point sont facultatifs si le module fait peu de communication et que ces communications se limitent à du contrôle et de la synchronisation comme le module Extractor. Les communications du module Extractor peuvent continuer à passer sur le bus OPB.

La dernière colonne du tableau 4.8 montre les liens point à point accélèrent l'exécution de la simulation. Non seulement, il y a moins de cycles à simuler par l'ordonnanceur SystemC mais aussi il y a moins de changements de contexte de processus à exécuter par l'ordonnanceur car les liens point à point ne font pas intervenir de processus. Tandis que les communications par bus sollicitent régulièrement les processus de l'arbitre et du temporisateur du bus OPB.

Conclusion :

La segmentation des modules dans un réseau multibus permet à certains modules de s'exécuter sans être perturbés par les communications abondantes d'un processeur. Dans une application de type *streaming*, l'utilisation de liens point à point avec des processeurs permet à ces derniers de mieux répondre au flux continu de données. L'utilisation de liens point à point entre un processeur et un module matériel est justifié pour un volume de données importantes et permet au processeur d'exécuter plus rapidement une communication avec un module matériel car ce médium n'est pas partagé.

4.3.2 Performance selon le mode d'arbitrage (phase 3 de la méthodologie)

Les résultats présentés dans cette section ont été effectués avec une configuration matérielle. Il n'y a pas de modules logiciels. Le seul paramètre configurable dans le modèle abstrait OPB est l'arbitrage. La taille du bus est fixe (32 bits) et le mode rafale associée au verrouillage du bus est géré par la librairie Space. Pour bien montrer l'effet de la configuration du protocole de communication du bus OPB sur l'optimisation d'un réseau multibus, le test a été effectué sur une **seule image 128*128 pixels** avec le placement des modules de la figure 4.8. La simulation est purement matérielle.

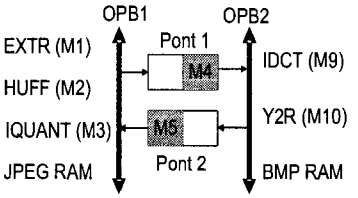

Figure 4.8 Topologie de test pour différents modes d'arbitrage

Le tableau 4.9 montre le nombre de cycles simulés en fonction de l'arbitrage du bus OPB1 et OPB2. En gras, apparaît la meilleure performance.

Tableau 4.9 Résultats des performances selon le mode d'arbitrage

Cas	Arbitrage OPB1[1]	Arbitrage OPB2[1]	Cycles	Nombre de *timeouts*[2]
1	Priorité M1>M2>M3>M4	Priorité M5>M6>M7	1 135 899	84 pour IDCT
2	LRU	LRU	1 133 109	0
3	LRU	Priorité M5>M6>M7	**1 132 992**	0
4	Priorité M4>M1>M2>M3	Priorité M5>M6>M7	1 134 931	0
5	Priorité M1>M2>M3>M4	Priorité M6>M7>M5	1 136 094	84 pour IDCT 9 pour IQUANT

[1] L'arbitrage FIFO n'est pas utilisé car il n'est pas présent dans l'IP matériel du bus OPB fournit par Xilinx

[2] Dus au communications inter-bus

Les résultats mettent en évidence l'influence de l'arbitrage dans un réseau multibus. Dans une architecture simple bus partagé, l'arbitrage n'a pas réellement d'influence sur la performance du système. Seule l'attribution du bus aux différents maitres change. En revanche, dans un réseau multibus utilisant des ponts OPB-OPB, l'arbitrage sur chaque bus change la dynamique d'attribution des bus mais modifie aussi la performance du système. Dans un réseau multibus OPB, la probabilité qu'un *timeout* se produise augmente avec l'utilisation des ponts et l'utilisation de l'arbitrage de plus haute priorité (la partie 3.4.2.2 décrit aussi le phénomène de famine).

Le tableau 4.9 montre que dans le Cas 1, 84 *timeouts* se produisent pour le module IDCT. Cela correspond aux acquittements Space envoyés par IDCT pour débloquer le module Iquant. Le Pont 2 étant le moins prioritaire sur OPB1, les communications intra-bus sur OPB1 et l'arbitrage empêchent le pont d'obtenir l'accès, ce qui entraine un nombre élevé de *timeouts* augmentant le nombre de cycles. Le même phénomène se répète pour le module IQUANT dans le cas 5. Pour remédier au problème, plusieurs solutions existent.

- Rendre les Pont 1 et 2 les plus prioritaires sur leur bus respectif. Ainsi, les

requêtes passant par le pont sont servies rapidement, ce que montre le cas 4 pour les *timeouts* de IDCT et le cas 1, 3 et 4 pour les *timeouts* de IQUANT.

- Utiliser un arbitrage dynamique comme le LRU qui permet un arbitrage plus juste. Les cas 2 et 3 montrent l'absence de *timeouts* pout IDCT car l'arbitrage LRU fait en sorte que le Pont 2 soit le plus prioritaire s'il n'a pas eu le bus depuis longtemps.

De plus, une combinaison de politique d'arbitrage dans un réseau multibus peut donner de bonne performance (cas 3).

En appliquant le même raisonnement pour le test avec **2 images de 128*128 pixels**, la configuration suivante du protocole du bus OPB donne les meilleurs résultats :

- **OPB1** : arbitrage de **plus haute priorité** avec dans l'ordre décroissant des priorités, **Pont OPB-OPB** > **Extractor** > **Huffman** > **Iquant** > **IDCT** > **Y2R**.
- **OPB2** : arbitrage de **plus haute priorité** avec dans l'ordre décroissant des priorités, **Edge**> **Face** > **Eye** > **Pont OPB-OPB**.

4.4 Risques liés à l'utilisation du pont direct

Lorsque l'on utilise un pont bidirectionnel direct pour relier deux bus, trois problèmes peuvent se rencontrer : le **blocage de deux bus**, la **famine** et **l'interblocage**.

Un pont direct ne stocke pas de données temporairement. Une donnée qui arrive est transmise sur l'autre bus dès que possible. Par conséquent, un module qui initie une transaction inter-bus **bloquera deux bus** pendant toute la durée du ou des transferts de données. Ce blocage accroît la surcharge de chaque bus et peut dégrader la performance de communication du système si les communications inter-bus sont

trop importantes. La configuration 'Multibus 2' (cf. tableau 4.6) montre un effet excessif du blocage de deux bus (graphe 4.6 de la partie 4.3.1.2). Les modules IDCT et Y2R écrivent et lisent la mémoire BITMAP sur l'autre bus. Les communications inter-bus sont très importantes. On obtient une performance de 3 160 841 cycles. En mettant la mémoire BITMAP double port, i.e. attaché au deux bus ('Multibus 2a' sur le graphe 4.6), on élimine les communications inter-bus par le pont OPB-OPB et la performance passe à 2 623 298 cycles. Ce gain a donc été obtenu en éliminant la surcharge apportée par le blocage de deux bus.

La **famine** est une situation dans laquelle une transaction est coincée dans le pont car l'interface maître de celui-ci n'obtient jamais l'accès à cause d'un maître plus prioritaire effectuant des communications intra-bus abondantes. La partie 4.3.2 montre un exemple de famine et comment le résoudre en changeant simplement le type d'arbitrage ou la priorité des maîtres.

Lors de l'exploration architecturale de l'application JPEG au niveau BCA, une des configurations multibus a mené à une situation d'interblocage[3]. Chaque bus OPB était configuré en mode arbitrage dynamique (LRU). La situation impliquée le module IDCT qui voulait écrire des données dans la mémoire BITMAP se trouvant sur l'autre bus OPB et le module Face qui envoyait un acquittement Space au module Extractor se trouvant sur l'autre bus OPB. Lorsque chacun de ces modules a obtenu l'accès au bus OPB, le *timeout* du bus géré par le temporisateur est déclenché. Chaque pont OPB-OPB doit obtenir l'accès au deuxième bus OPB avant le 16ème cycle. Si le deuxième bus OPB est accordé au pont, ce dernier signale au temporisateur du premier bus OPB d'inhiber le *timeout*. Sinon, la transaction en cours est interrompue et chaque module doit se faire réarbitrer pour relancer sa transaction. Cette situation peut donc se répéter indéfiniment.

[3]L'interblocage est expliqué dans la partie 3.4.2.2

Un moyen de casser l'interblocage est de changer le type d'arbitrage. Ceci permet de changer la dynamique d'attribution des bus OPB et d'éviter la situation de communications inter-bus dans le même cycle. Une autre solution consiste à activer le mode anti-interblocage du modèle abstrait du pont OPB-OPB. En effet, lorsque deux ponts OPB-OPB sont utilisés pour relier deux bus OPB, ils sont connectés entre-eux pour savoir si une situation d'interblocage a lieu (cf. partie 3.4.2.2 pour la détection d'interblocage). Si l'interblocage est décelé à la suite de un ou plusieurs *timeouts* alors un des ponts OPB-OPB voit sa transaction interrompue immédiatement alors que le deuxième pont attend un délai de 10 cycles avant d'interrompre sa transaction. Ceci donne la chance à l'autre transaction de passer sur les deux bus. La figure 4.9 résume le mode anti-interblocage.

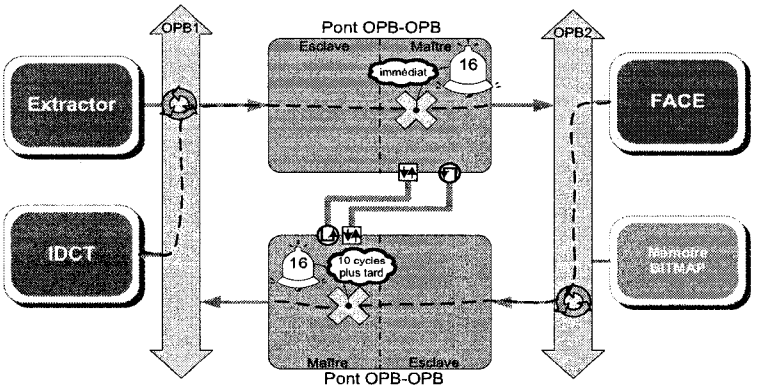

Figure 4.9 Risques liés à l'utilisation d'un pont

Le tableau 4.10 présente le nombre de cycles simulés pour casser l'interblocage avec trois méthodes différentes. Le changement d'arbitrage (de LRU à plus haute priorité), l'arbitrage LRU avec le mode anti-interblocage et l'utilisation de la mémoire double port permettent l'exécution complète de l'application. Dans ce cas-ci, le mode anti-interblocage apporte une performance améliorée pour le système mais il est parfois plus judicieux de limiter les communications inter-bus en analysant les modules à l'origine de l'interblocage. Ainsi, comme le montre l'utilisation de la mémoire double port, on élimine beaucoup de communications inter-bus dues au module IDCT

diminuant donc le risque d'interblocage.

Tableau 4.10 Performance des methodes anti-interblocage

Arbitrage de plus haute priorité sur OPB1 et OPB2	3 160 841
Mode anti-interblocage actif (arbitrage LRU sur OPB1 et OPB2)	3 082 508
Mémoire BITMAP double port	2 623 298

CONCLUSION ET TRAVAUX FUTURS

L'association de plusieurs processeurs sur une puce a donc donné naissance aux systèmes-sur-puce multiprocesseurs (MPSoC). Les MPSoCs introduisent une nouvelle dimension dans la conception des systèmes-sur-puce, le parallélisme. Ce dernier offre de multiples avantages. Il permet d'intégrer plus de fonctions dans un système, plusieurs algorithmes peuvent s'exécuter en parallèle. Le parallélisme entraine donc l'exécution de multiples instructions par cycle et par conséquent le traitement de multiples données. Aujourd'hui, la plupart des plateformes multimédia sont déjà des MPSoCs. Ces types de plateforme sont caractérisés par un volume de données importantes échangées et traitées par les processeurs. Pour satisfaire le trafic généré par des applications multimédias, les processeurs doivent être reliés par un réseau de communication performant.

L'exploration des réseaux de communication accompagne la complexité croissante des MPSoCs. Choisir le bon réseau de communication, le configurer correctement et placer les composants au sein de ce réseau pour obtenir la meilleure performance possible peut s'avérer un vrai casse-tête. La popularité de la méthodologie ESL pour concevoir des modèles abstraits de composants permet d'explorer les réseaux de communication au niveau système. En s'élevant dans les niveaux d'abstraction, notre compréhension du système est facilitée. Il en résulte une meilleure analyse et implémentation d'un réseau de communication. La plateforme Space, développé par le laboratoire de Codesign de l'École Polytechnique de Montréal, offre ce type de méthodologie au niveau système.

À partir de l'assemblage de composants IP, la plateforme Space permet facilement de créer un système. À travers ses trois niveaux d'abstraction, Space offre la possibilité d'explorer divers types d'architectures de communication. Ces dernières se divisent en

trois grandes catégories : les bus, les liens point à point et les réseaux-sur-puce comme le décrit le chapitre 2. Space permet de construire et d'évaluer des architectures de communication basées sur les bus et les liens point à point.

Des composants abstraits ont donc été développés ou améliorés à partir du langage de modélisation SystemC afin de construire des architectures de communication à haut niveau. Ainsi, au niveau TF de la plateforme Space, le canal de communication TF permet de reproduire le fonctionnement général de certains bus standards tels que les bus OPB et PLB du protocole CoreConnect d'IBM et les bus AHB, APB et ASB du protocole AMBA de ARM. Grâce à différents paramètres (taille du bus, type d'arbitrage, latence des diverses phases d'un transfert d'une donnée, mode rafale), le canal TF se configure aisément pour reproduire le comportement d'un bus. Pour former un réseau multibus, un pont fonctionnel est disponible. Il offre les deux grandes fonctionnalités d'un pont reliant deux bus : le mode Direct et Stocke-et-Transfert. Le premier mode permet le transfert direct de la donnée et le deuxième, le transfert différé grâce à l'utilisation d'une FIFO. En mode Stocke-et-Transfert, le pont adapte deux domaines d'horloge différents très répandus dans les bus hiérarchiques. Au niveau BCA, le modèle abstrait du bus OPB déjà disponible a été complété par l'ajout d'un arbitrage dynamique, d'un *timeout* et par la modification du protocole afin de rendre le bus OPB utilisable dans un réseau multibus. Le pont OPB-OPB a été aussi créé pour relier deux bus OPB. Il fonctionne uniquement en mode direct. Une paire de ponts OPB-OPB possèdent des mécanismes de détection d'interblocage et de famine ainsi qu'un mode anti-interblocage.

Les bases d'un concept de lien point à point générique utilisable dans les différents niveaux de Space ont été proposées. Ces liens peuvent reliés un module matériel à un ou plusieurs modules matériels et un ou plusieurs modules matériels à un ou plusieurs modules logicielles.

Afin d'explorer les nombreuses possibilités d'architectures de communication offertes par l'assemblage des composants précédents, une méthode d'exploration des communications-sur-puce multi-niveaux d'abstraction a été proposée. Au niveau TF, la méthode a pour but d'améliorer une architecture à bus partagé simple avec un réseau multibus dans lequel les modules sont placés selon en combinant une techniques de regroupement des communications et de concurrence. Au niveau BCA, le réseau multibus retenu est optimisé en configurant le protocole des différents bus de façon adéquat et/ou en utilisant des composants annexes tels que les liens point à point et des mémoires double port. Il est alors possible d'obtenir une architecture multibus optimisée pour l'application choisie avec le bon partitionnement matériel/logiciel qui convient.

Les résultats montrent que la simulation d'une architecture de communication au niveau TF offre une accélération en moyenne de 2,6 et une estimation des communications supérieure à 80% par rapport à une simulation au niveau BCA dans le cas d'un trafic généré par l'application JPEG. Un trafic idéal dévoile que l'accélération apportée par le modèle d'architecture de communication au niveau TF dépend de la taille de la transaction. E.g. l'accélération atteint 4,3 pour des transactions de 17 mots et oscille autour de 2 pour des transactions de 1 mot. De plus, la méthode d'exploration architecturale appliquée sur une décompression JPEG avec diverses détections de formes démontre le gain de performance obtenu grâce à un réseau multibus utilisant des ponts face à un simple bus partagé. En effet, l'utilisation d'un pont permet de répartir la contention d'un seul bus sur deux bus en plaçant judicieusement les modules sur chaque bus. Dans l'application JPEG, une certaine répartition des modules sur deux bus a permis d'obtenir une amélioration de la performance de 14,6% en termes de cycles. Toutefois, les résultats prouvent aussi que la performance d'un réseau multibus dépend des communications intra- et inter-bus (une dégradation de la performance pouvant atteindre 50% a été observée). Par

ailleurs, les tests montrent aussi que l'utilisation de liens point à point permet d'alléger la contention sur le bus en créant des chemins spécifiques entre les modules. Les liens point à point permettent à un processeur d'accélérer grandement ses communications. Une accélération comprise entre 1,8 et 4,3 a été constatée pour les tests. De la même façon, l'utilisation d'une mémoire double port améliore le parallélisme d'un réseau multibus par des accès simultanés (e.g. performance améliorée de 20%). Et enfin, la méthode d'exploration a mis en évidence les dangers rencontrés lors de l'utilisation de ponts, notamment le phénomène d'interblocage et de famine.

Travaux futurs

Dans ce projet, les architectures multibus testés ont porté sur des architectures homogènes de bus (bus OPB) et de processeurs (MicroBlaze). Le bus OPB reste un bus de périphériques avec une performance moyenne. Sur le FPGA Virtex-II Pro de Xilinx, il est souvent utilisé à une fréquence de 50MHz avec une largeur de 32 bits, soit une bande passante théorique de 200 Mo/s. Un seul bus OPB ne permet pas de construire un système multiprocesseur efficace. Deux processeurs sur le même bus OPB ne permettraient pas d'accomplir de bonne performance car le bus ne pourrait satisfaire assez rapidement toutes les communications. De plus, l'utilisation du bus OPB et du pont OPB-OPB permettent seulement de construire des systèmes multiprocesseurs homogènes à base de MicroBlaze. Le pont OPB-OPB reste un goulot d'étranglement pour des communications inter-processeurs. Une autre possibilité offerte par les FPGAs de Xilinx est l'utilisation de processeurs PowerPC d'IBM et du bus PLB qui pourrait conduire à étudier à haut niveau des architectures multibus hiérarchiques hétérogènes composées de processeurs MicroBlaze et PowerPC et des bus PLB et OPB. À l'heure d'écriture du mémoire, le modèle SystemC du processeur PowerPC 405 d'IBM est disponible et le modèle abstrait du bus PLB est en cours de développement.

Ce mémoire pourrait donc constituer une base pour étudier des architectures

hétérogènes de bus et de processeurs. La méthode d'exploration pourrait être testée et validée pour obtenir une architecture de communication optimisée avec les bus OPB et PLB. Dans le cas de bus hiérarchiques, les problèmes liés à l'adaptation de domaines d'horloge différents à travers les ponts pourraient être abordés. Pour les étudier, les modèles au niveau BCA des ponts PLB-OPB et OPB-PLB devront être développés. Leur développement pourrait s'appuyer sur le modèle du pont OPB-OPB et du pont fonctionnel en mode Stocke-et-Transfert. Il serait ainsi faisable de simuler le type d'architecture présentée dans la figure 4.10.

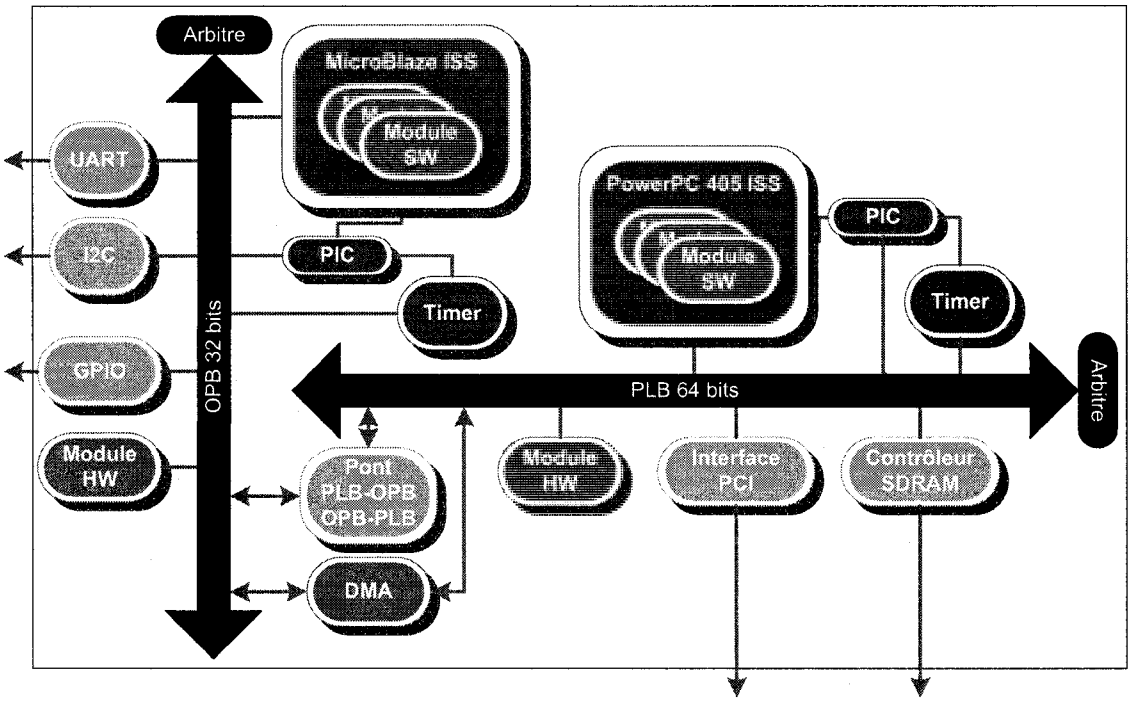

Figure 4.10 Architecture typique PowerPC-CoreConnect

Les bus standards haute performance comme le bus PLB du protocole CoreConnect offre une architecture pipelinée [31] pour augmenter la bande passante possible. Ce pipeline est possible grâce à un bus de données séparé en bus d'écriture et de lecture et à un bus d'adresse. De plus, le pipeline est implémenté aussi dans l'arbitre du PLB pour permettre de traiter les requêtes plus rapidement. Ainsi, les cycles d'adresse peuvent se chevaucher avec des cycles de transfert de données en écriture et en lecture

et les cycles de transfert de données en lecture avec peuvent se chevaucher avec les cycles de transfert de données en écriture. Le pipeline d'adresse permet à un nouveau transfert de commencer alors que le transfert courant n'a pas fini.

Le modèle du canal TF pourrait donc être amélioré afin de simuler l'aspect pipeline du bus PLB. Cela consisterait à employer des processus SystemC dans le canal TF et dans l'arbitre et de synchroniser le tout par des évènements notifiés et attendus aux bons endroits.

RÉFÉRENCES

[1] (2006). SoC Interconnection: Wishbone. Consulté le 23-09-2007, http://www.opencores.org.uk/projects.cgi/web/wishbone.

[2] Ahmadinia, A., Bobda, C., Ding, J., Majer, M., Teich, J., Fekete, S. P. et van der Veen, J. C. (2005). A practical approach for circuit routing on dynamic reconfigurable devices. *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on.* (pp. 84–90).

[3] Altera, (2007). Embedded Processors. Consulté le 19-09-2007, http://www.altera.com/products/ip/processors/ipm-index.jsp.

[4] ARM, (1999). AMBA Specification (Rev 2.0). Rapport technique.

[5] ARM, (2007). AMBA Overview. Consulté le 23-09-2007, http://www.arm.com/products/solutions/AMBAHomePage.html.

[6] ARM, (2007). Processor Overview. Consulté le 19-09-2007, http://www.arm.com/products/CPUs.

[7] ARM, (2007). RealView Core Generator. Consulté le 21-09-2007, http://www.arm.com/products/DevTools/CoreGenerator.html.

[8] Arteris, (2005). A comparison of Network-on-Chip and Busses. Consulté le 19-09-2007, http://www.us.design-reuse.com/articles/article10496.html.

[9] Bartic, T. A., Mignolet, J. Y., Nollet, V., Marescaux, T., Verkest, D., Vernalde, S. et Lauwereins, R., (2005). Topology adaptive network-on-chip design and implementation. *Computers and Digital Techniques, IEE Proceedings -, 152*(4), 467–472.

[10] Benini, L. et Bertozzi, D., (2005). Network-on-chip architectures and design methods. *Computers and Digital Techniques, IEE Proceedings* -, *152*(2), 261–272.

[11] Bluespec, (2005). Bluespec SystemVerilog. Consulté le 17-09-2007, http://www.bluespec.com/products/bsv.htm.

[12] Cai, L. et Gajski, D., (2003). Transaction Level Modeling in System Level Design. Consulté le 23-09-2007. www.cecs.uci.edu/technical_report/TR03-10.pdf.

[13] Carloni, L. P. et SangiovanniVincentelli, A. L. (2003). On Chip Communication Design: Roadblocks and Avenues. *CODES+ISSS*. Newport Beach, California, USA.

[14] Celoxica, (2007). DK Design Suite. Consulté le 16-09-2007, http://www.celoxica.com/products/dk/default.asp.

[15] Cheng-Ta, H. et Pedram, M., (2002). Architectural energy optimization by bus splitting. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *21*(4), 408–414.

[16] Chevalier, J., de Nanclas, M., Filion, L., Benny, O., Rondonneau, M., Bois, G. et El Mostapha, A., (2006). A SystemC refinement methodology for embedded software. *Design and Test of Computers, IEEE*, *23*(2), 148–158.

[17] Claasen, T. A. C. M., (2006). An industry perspective on current and future state of the art in system-on-chip (SoC) technology. *Proceedings of the IEEE*, *94*(6), 1121–1137.

[18] CoFluent. CoFluent Studio. Consulté le 21-09-2007, http://www.cofluentdesign.com/pageLibre00010079.html.

[19] Darringer, J. A., Bergamaschi, R. A., Bhattacharya, S., Brand, D., Herkersdorf, A., Morrell, J. K., Nair, I. I., Sagmeister, P. et Shin, Y., (2002). Early analysis tools for system-on-a-chip design.

[20] Densmore, D. et Passerone, R., (2006). A Platform-Based Taxonomy for ESL Design. *Design and Test of Computers, IEEE*, *23*(5), 359–374.

[21] Dutt, N., Dutt, N. et Kiyoung, C., (2003). Configurable processors for embedded computing Configurable processors for embedded computing. *Computer*, *36*(1), 120–123. 0018-9162.

[22] Dutt, N., Lahiri, K. et Pasricha, S. (2007). Modelling, Analysis and Design of Bus-based SoC Communication Architectures. *DATE*. Nice, France. consulté le 22-09-2007 http://www.date-conference.com/conference/2007/prog/progdetail.php?progid=G1.

[23] Filion, L., (2006). Documentation du modele transactionnel OPB intégré dans SPACE. Rapport technique.

[24] Ganssle, J., (2006). What processor is in your product? Consulté le 19-09-2007, www.embedded.com.

[25] Gerin, P., Sungjoo, Y., Nicolescu, G. et Jerraya, A. A. (2001). Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific.* (pp. 63–68).

[26] Goyette, S., (2007). Elaboration d'un modèle d'abstraction des communications point-à-point en considérant une architecture multiprocesseur hétérogène sur puce. Rapport technique.

[27] Haverinen, A., Leclercq, M., Weyrich, N. et Wingard, D., (2002). SystemC based SoC Communication Modeling for the OCP Protocol. Consulté le 16-09-2007, http://www.ocpip.org/socket/whitepapers.

[28] Hubner, M., Paulsson, K. et Becker, J. (2005). Parallel and Flexible Multiprocessor System-On-Chip for Adaptive Automotive Applications based on Xilinx MicroBlaze Soft-Cores. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International.* (pp. 149a–149a).

[29] IBM. CoreConnect Bus Architecture. Consulté le 23-09-2007, http://www-01.ibm.com/chips/techlib/techlib.nsf/pages/main.

[30] IBM, (2001). On-Chip Peripheral Bus, Architecture Specifications, Version 2.1. Rapport technique.

[31] IBM, (2004). 128-bit Processor Local Bus, Architecture Specifications, Version 4.6. Rapport technique.

[32] IBM, (2007). Power Architecture Solutions Brochure. Consulté le 19-09-2007, http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs.

[33] Instruments, N., (2007). ESL Explained. Consultée le 12-09-2007, http://zone.ni.com/devzone/cda/pub/p/id/142.

[34] Instruments, T. High-Performance: OMAP3430. Consulté le 19-09-2007, http://focus.ti.com.

[35] Instruments, T., (2007). Digital Signal Processing. Consulté le 19-09-2007, http://focus.ti.com/dsp/docs/dsphome.tsp?sectionId=46.

[36] Jerraya, A. et Wolf, W., (2005). Architecture of Embedded Microprocessors. *Multiprocessor Systems-on-Chips*, Morgan Kaufmann. pp. 81–110.

[37] Jerraya, A. et Wolf, W., (2005). Chapter 1.7 - Software. *Multiprocessor Systems-on-Chips*, Morgan Kaufmann. pp. 14–18.

[38] Jerraya, A. et Wolf, W., (2005). Chapter 2.3 - Energy-aware memory system design. *Multiprocessor Systems-on-Chips*, Morgan Kaufmann. pp. 27–34.

[39] Jerraya, A. et Wolf, W., (2005). Chapter 7.3 - system-level analysis for designing communication architectures. *Multiprocessor Systems-on-Chips*, Morgan Kaufmann. pp. 194–203.

[40] Jerraya, A. et Wolf, W., (2005). Design of Communication Architectures for High-Performance and Energy-Efficient Systems-on-Chips. *Multiprocessor Systems-on-Chips*, Morgan Kaufmann. pp. 187–250.

[41] Kirovski, D. et Potkonjak, M. (1997). System-level Synthesis Of Low-power Hard Real-time Systems. *Design Automation Conference, 1997. Proceedings of the 34th.* (pp. 697–702).

[42] Klingauf, W., Burton, M., Günzel, R. et Golze, U., (2007). Why We Need Standards for Transaction-Level Modeling. Consulté le 15-09-2007, www.soccentral.com.

[43] Knudsen, P. V. et Madsen, J., (1999). Integrating communication protocol selection with hardware/software codesign. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *18*(8), 1077–1095.

[44] Kyeong Keol, R., Eung, S. et Mooney, V. J. (2001). A comparison of five different multiprocessor SoC bus architectures. *Digital Systems, Design, 2001. Proceedings. Euromicro Symposium on.* (pp. 202–209).

[45] Kyeong Keol, R. et Mooney, V. J., I., (2004). Automated bus generation for multiprocessor SoC design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *23*(11), 1531–1549.

[46] Lahiri, K., Raghunathan, A. et Dey, S., (2001). System-level performance analysis for designing on-chip communication architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *20*(6), 768–783.

[47] Lahiri, K., Raghunathan, A. et Dey, S., (2004). Design space exploration for optimizing on-chip communication architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, *23*(6), 952–961.

[48] Leibson, S., (2005). The end of Moore's law. Consulté le 14-09-2007. www.embedded.com.

[49] Linux, (2006). Snapshot of the embedded Linux market. Consulté le 19-09-2007. http://www.linuxdevices.com/articles/AT7070519787.html.

[50] Loghi, M., Angiolini, F., Bertozzi, D., Benini, L. et Zafalon, R. (2004). Analyzing on-chip communication in a MPSoC environment. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings.* (Vol. 2, pp. 752–757 Vol.2).

[51] Lysecky, R. et Vahid, F. (2005). A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning. *Design, Automation and Test in Europe, 2005. Proceedings.* (pp. 18–23 Vol. 1).

[52] Martin, G. (2006). Overview of the MPSoC design challenge. *Design Automation Conference, 2006 43rd ACM/IEEE.* (pp. 274–279).

[53] Martin, G., (2007). Commentary: A new definition of ESL. Consulté le 14-09-2007, www.eetimes.com.

[54] Migliorini, C., Goyette, S. et Filion, L., (2007). Spécification du lien point à point Elix, Simtek, GenX. Rapport technique.

[55] MIPS, (2007). MIPS. Architectures. Consulté le 19-09-2007, http://www.mips.com/products/architectures.

[56] Moss, L., (2009). Profilage, caractérisation et partitionnement fonctionnel dans une plateforme de conception de systèmes embarqués. Rapport technique.

[57] Moss, L., de Nanclas, M., Filion, L., Fontaine, S., Bois, G. et Aboulhamid, M. (2007). Seamless Hardware/Software Performance Co-Monitoring in a Codesign Simulation Environment with RTOS Support. *Design, Automation and Test in Europe Conference and Exhibition, 2007. DATE '07.* (pp. 1–6).

[58] Natick, (2007). The Electronic System Level (ESL) Tools Market: Virtual System Prototyping/Simulation Tools Predicted to Grow Fastest. Consulté le 15-09-2007, www.us.design-reuse.com.

[59] OSCI, (2007). SystemC. Consulté le 16-09-2007, www.systemc.org.

[60] PalmChip, (2003). Overview of CoreFrame Architecture. Consulté le 23-09-2007, http://www.palmchipasp.com/coreframe_downloads.asp.

[61] Palmer, M., (2002). Two New CoWare Products Provide Increased Productivity for Cross-Functional SoC Design Teams. Consulté 21-09-2007, http://www.coware.com/news/press404.htm.

[62] Park, G.-H. et al., (2006). Architecture Exploration and Performance Verification Environments for a Multi-Core Embedded SOC Platform Design. Consulté le 19-09-2007, http://cag.csail.mit.edu/warfp2006/submissions/park_samsungelectronics.pdf.

[63] Pasricha, S., Dutt, N. et Ben-Romdhane, M. (2006). Constraint-driven bus matrix synthesis for MPSoC. *Design Automation, 2006. Asia and South Pacific Conference on.* (p. 6 pp.).

[64] Perrier, V., (2007). System architecting by prospective performances analysis and architectural exploration. Consulté 21-09-2007,http://www.us.design-reuse.com/articles/article7251.html.

[65] Philips. Nexperia. Consulté le 19-09-2007, www.nxp.com.

[66] Qualis, (2000). Designing with reuse in mind. Consulté le 14-09-2007. www.smithediting.com/samples/reuse.pdf.

[67] Ravindran, K., Satish, N., Yujia, J. et Keutzer, K. (2005). An FPGA-based soft multiprocessor system for IPv4 packet forwarding. *Field Programmable Logic and Applications, 2005. International Conference on.* (pp. 487–492).

[68] Rizzatti, L. et Schutten, R., (2007). Defining the TLM-to-RTL Design Flow. Consulté le 15-09-2007, www.embedded.com.

[69] STMicroelectronics. Mobile multimedia application processor - Nomadik. Consulté le 19-09-2007 http://www.st.com/stonline/products/literature/bd/13598/stn8811a12.pdf.

[70] STMicroelectronics. STBus Interconnect. Consulté le 23-09-2007, http://www.st.com/stonline/products/technologies/soc/stbus.htm.

[71] Swan, S., (2006). A Tutorial Introduction to the SystemC TLM Standard. Consulté le 16-09-2009, www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-13-OSCI_2_swan.pdf.

[72] Synopsys, (2007). DesignWare SystemC Libraries. Consulté le 21-09-2007, http://www.synopsys.com/products/designware/docs/toc/dwlibdocs.php.

[73] SystemC, (2007). About OSCI. Consulté le 14-09-2007, www.systemc.org/about.

[74] Tensilica. Xtensa Xplorer Integrated Design Environment (IDE). Consulté le 19-09-2007, http://www.tensilica.com/products/sw_sw_xplorer.htm.

[75] Terrence, S. T. M., Pete, S., Peter, Y. K. C. et Wayne, L. (2006). On-FPGA Communication Architectures and Design Factors. (pp. 1–8).

[76] Tsasakou, S., Voros, N. S., Koziotis, M., Verkest, D., Prayati, A. et Birbas, A. (1999). Hardware-software co-design of embedded systems using CoWare's N2C methodology for application development. *Electronics, Circuits and Systems, 1999. Proceedings of ICECS '99. The 6th IEEE International Conference on.* (Vol. 1, pp. 59–62 vol.1).

[77] VaST. CoMET. Consulté le 21-09-2007. http://www.vastsystems.com/docs/CoMET_mar2007.pdf.

[78] Viteau, V., (2006). TLM Abstration Levels. Consulté le 17-09-2007, http://www.systemc.org/Discussion_Forums.

[79] Wingard, D., (1999). Sonics SOC Integration Architecture. Consulté le 23-09-2007. http://grouper.ieee.org/groups/1500/jan99/Sonics.pdf.

[80] Wolf, W. (2004). The future of multiprocessor systems-on-chips. *Design Automation Conference, 2004. Proceedings. 41st.* (pp. 681–685).

[81] Xilinx, (2004). OPB to PLB Bridge (v1.00c). Rapport technique.

[82] Xilinx, (2005). Fast Simplex Link (FSL) Bus (v2.00a). Rapport technique.

[83] Xilinx, (2005). On-Chip Peripheral Bus V2.0 with OPB Arbiter (v1.10c). Rapport technique.

[84] Xilinx, (2005). OPB to OPB Bridge (Lite Version) (v1.00a). Rapport technique.

[85] Xilinx, (2005). PLB to OPB Bridge (v1.01a). Rapport technique.

[86] Xilinx, (2007). MicroBlaze Architecture. Consulté le 19-09-2007, http://www.xilinx.com/ipcenter/processor_central/microblaze/architecture.htm.

[87] Zeidman, B., (2005). Back to the basics: Programmable Systems on a Chip. Consulté le 22-09-2007. http://www.embedded.com.

ANNEXE I

SYSTEMC

Les principaux éléments :

La plateforme SPACE utilise SystemC qui est une librairie de classes C++ pour la modélisation de systèmes logiciels/matériels. SystemC fournit les éléments essentiels à la modélisation au niveau système : les modules, les ports avec fonctionnalités étendues, les évènements, la sensibilité dynamique, les interfaces et les canaux définis par l'utilisateur, et bien plus. Voici une brève description des caractéristiques intéressantes.

Tout design en SystemC est avant tout constitué de **MODULES**, l'élément structurel de base d'un système. Le module sert à partitionner un design complexe en sous-blocs qui regroupent des systèmes internes, qui sont plus simples à manipuler, à tester et à vérifier. Le module contient normalement des ports pour la communication avec l'extérieur, des signaux d'activation locaux, des variables locales ou tampons, des processus et un constructeur du module. Le module peut également être composé de modules internes (ou modules hiérarchiques).

Les **PROCESSUS** sont utilisés pour décrire un traitement à exécuter. Le processus SystemC (*process*) est similaire au processus en VHDL ou au module en Verilog. Les processus ont généralement une liste de sensibilités qui leur permettent d'être activés pour exécution. Le processus apporte les mécanismes nécessaires pour une simulation parallèle (*concurrent*). Les modules peuvent ainsi contenir un ou plusieurs processus. Ils peuvent être perpétuels (SC_ METHOD), perpétuels synchrones (SC_CTHREAD) ou momentanés (SC_ THREAD). Tous les processus décrivent

des exécutions séquentielles. Les SC_CTHREAD et SC_ THREAD peuvent être interrompus et réactivés, tandis que les SC_METHOD s'activent, s'exécutent et se terminent sans interruption.

Le **PORT** permet de faire entrer ou sortir des informations d'un module. En SystemC, le port se connecte soit à l'interface d'un canal de transmission ou à un port hiérarchique parent. Les signaux traditionnellement utilisés pour connecter les ports sont aussi des canaux de communication. SystemC offre des ports d'entrée (*sc_in<T>*) les ports de sortie (*sc_out<T>*) et les ports bidirectionnels (*sc_inout<T>*). Le T entre les crochets du port représente le type du port en question. Les ports SystemC sont donc implantés grâce aux gabarits de type (*templates*) C++. Seuls les ports de mêmes types peuvent être connectés entre eux. Il est possible de lire la valeur d'un port par la méthode *read()* et d'écrire dans un port par la méthode *write()*. Un port est donc associé à une interface qui contient un certain nombre de méthodes. Ces méthodes sont donc utilisables par le module en y accédant par le port.

L'HORLOGE est un signal spécial sous SystemC. Les horloges conservent l'information sur le temps qui passe au cours d'une simulation. Les horloges servent également à la synchronisation des processus. L'horloge, de type *sc_clock*, peut être ajustée au choix du programmeur selon sa période, sa répartition haut/bas, etc. Les ports particuliers sont nécessaires pour brancher les horloges, il s'agit des *sc_in_clk* et des *sc_out_clk*.

L'INTERFACE est un bloc purement fonctionnel et abstrait définissant un ensemble de méthodes, sans implantation ni variable. Les méthodes de ces interfaces sont implantées dans les canaux. Un canal peut implanter une ou plusieurs interfaces, qu'il veut bien supporter. Le port communique ainsi par l'intermédiaire des méthodes de l'interface implantées par le canal.

Les **CANAUX** sont reliés aux ports comme nous l'avons dit. Ce sont les canaux qui

activeront les ports sensibles d'un module. SystemC propose deux types de canaux. La première catégorie, les canaux élémentaires, ne contiennent aucun processus ou autre élément structurel définis par l'usager. Ils sont inclus à même la bibliothèque SystemC. Ils sont, par exemple, le *sc_signal* qui active les ports d'entrée sur un changement de la valeur du canal, *sc_buffer* qui active les ports d'entrée sur toute écriture et *sc_fifo* qui permet d'emmagasiner plusieurs valeurs dans une file de type premier arrivé premier sorti (*first in first out*). Il est également possible de créer ses propres canaux, plus complexes et qui définissent une interface configurable. Il s'agit de canaux hiérarchiques : ces canaux sont en fait des modules, puisqu'ils possèdent une structure et des processus. On peut vouloir décrire ses propres canaux pour spécifier un protocole précis ou encore une norme de bus.

Les listes de **SENSIBILITES** définies dans SystemC sont statiques, une fois définies, elles ne peuvent changer au cours de la simulation. Cependant, il est possible que l'on veuille ajouter dynamiquement et occasionnellement, au cours de la simulation, certains événements dans la liste de sensibilité. L'objet *sc_event* offre cette possibilité. Plusieurs options sont inclues avec ces objets, notamment pour l'attente d'un ou plusieurs événements simultanés ou un nombre de cycles prédéterminé (grâce à l'objet *sc_time*). On contrôle les événements à l'aide de méthodes de notification (*notify*) et d'attente (*wait*).

La figure I.1 montre la notation graphique utilisée pour SystemC.

Les phases de simulation :

Une implémentation de la librairie de classe SystemC inclut un ensemble public (*shell*) de macros, fonctions et classes prédéfinies directement utilisables par l'application. Elle inclut aussi un ensemble privée qui définit les sémantiques du noyau de SystemC. L'exécution d'une application SystemC comprend une phase **d'élaboration** suivie par une phase de **simulation**. L'élaboration permet la création d'une hiérarchie de

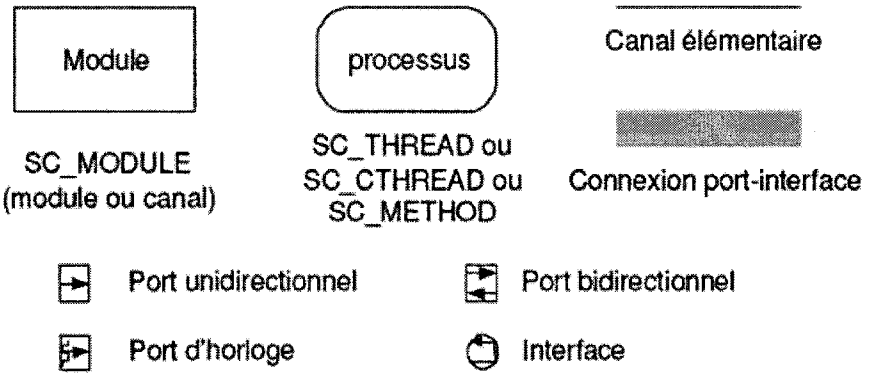

Figure I.1 Convention des représentations graphiques de SystemC

module. L'élaboration se compose de l'exécution du code de l'application, du *shell* et du code du noyau. La simulation implique l'exécution de **l'ordonnanceur** qui gère l'exécution des **processus** de l'application.

Les phases d'élaboration et de simulation suivent la séquence suivante :

1. Élaboration : construction de la hiérarchie des modules
2. Élaboration : procédure de rappel (*callback*) **before_end_of_elaboration()**
3. Élaboration : procédure de rappel (*callback*) **end_of_elaboration()**
4. Simulation : procédure de rappel (*callback*) **start_of_simulation()**
5. Simulation : phase d'initialisation
6. Simulation : phase d'évaluation, de mise à jour, de notification retardée d'un delta-cycle et/ou d'un temps.
7. Simulation : destruction de la hiérarchie des modules

L'élaboration créée donc les structures internes de données requis par le noyau pour lancer la simulation. La hiérarchie de module (modules, ports, canaux primaires et processus) est créée et les ports sont connectés aux canaux.

La simulation consiste en l'exécution de l'ordonnanceur. Celui-ci a pour tâche principale de déclencher et suspendre l'exécution des processus fournit par l'utilisateur. L'ordonnanceur est dirigé par les **évènements**, i.e. les processus sont exécutés en réponse à l'occurrence d'un évènement. Les évènements se déclenchent (sont **notifiés**) à des moments précis dans le temps de simulation. Ils sont représentés par des objets de classe **sc_event**. Le temps de simulation est une quantité entière initialisé à zéro au début de la simulation et incrémenté pendant la simulation. La signification physique de la valeur entière représentant le temps dans le noyau est déterminée par la résolution du temps de simulation. Le temps et les intervalles sont représentés par la classe **sc_time**. Certaines fonctions permettent d'exprimer le temps sous la forme valeur **double**, classe **sc_time_unit**.

L'ordonnanceur de SystemC :

L'ordonnanceur (***scheduler***) de SystemC détermine l'ordre d'exécution des processus. Cet ordre d'exécution est régit par les **listes de sensibilités** des ports, mais aussi par les **notifications d'événements** qui se produisent dans le système. La simulation s'arrête dès qu'il n'y a plus d'évènements ou dès l'appel explicite à une fonction d'arrêt (fonction *sc_stop*).

Pour la modélisation de matériel, le simulateur SystemC supporte le concept de **delta-cycle**, également utilisé en VHDL. Un delta-cycle est une unité de temps infinitésimale qui permet de créer un ordre partiel d'exécution entre les cycles d'horloges fixes. Un delta-cycle est composé de deux phases distinctes : une phase d'**évaluation** et une phase de **mise à jour** (**evaluate-update**). Ainsi, le simulateur exécute tous les processus prêts avant de mettre les sorties à jour ou d'incrémenter la période d'horloge. Il est possible que plusieurs itérations (delta-cycles) se produisent entre deux coups d'horloge.

SystemC supporte également des notifications temporisées ou non, qui causeront

l'exécution d'un processus après un intervalle de temps spécifié en argument ou immédiatement, s'il n'y a pas d'argument.

Le simulateur parcourt les 6 étapes suivantes lors de toute simulation :

1. Incrémentation des horloges.
2. Exécution des SC_METHOD/SC_THREAD qui ont un changement à leur entrée.
3. Mise à jour des sorties des processus de type SC_CTHREAD. Leur exécution se fera à l'étape 5.
4. Si des changements surviennent sur une ou plusieurs sorties, recommencer à partir de l'étape 2.
5. Exécution des SC_CTHREAD. Les sorties seront propagées à l'étape 3 du prochain coup d'horloge.
6. Incrémentation du temps de simulation et redémarrage à l'étape 1.

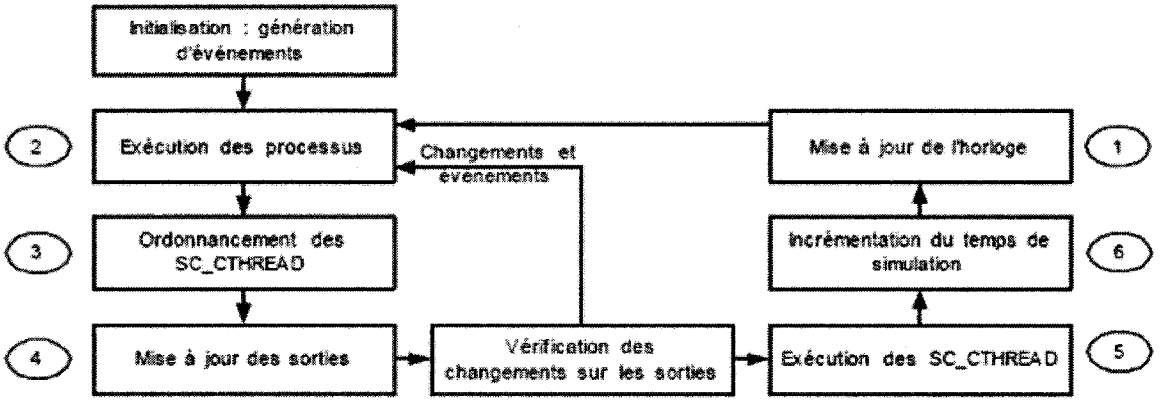

Figure I.2 Ordonnanceur de SystemC

Le schéma I.2 schématise les étapes expliquées ci-dessus. Il est important de mentionner que tous les processus, sauf ceux de type SC_CTHREAD, sont exécutés durant la phase d'**initialisation**. Chaque processus est exécuté une fois durant

l'initialisation et est exécuté jusqu'à ce qu'un **wait()** soit rencontré. Il est possible cependant d'empêcher l'initialisation de tel ou tel processus à l'aide de la méthode *dont_initialize()* qui se déclare dans le constructeur après des processus SC_METHOD ou des SC_THREAD. Un processus qui n'est pas initialisé n'est pas prêt à exécuter. Ce qui veut dire que le processus commence son exécution dès qu'il est déclenché par le premier événement.

La notification des évènements :

Un événement peut être notifié de 3 façons grâce à la méthode **notify()**. La notification **immédiate** permet le déclenchement de l'événement durant la phase évaluation du delta-cycle. Ceci se fait par la commande *notify* sans argument : **notify()** La notification **retardé d'un delta-cycle** permet le déclenchement de l'événement durant la phase évaluation du prochain delta-cycle. Ceci se fait par la commande *notify* avec argument 0 ou SC_ ZERO_TIME : **notify(0, SC_NS)** ou **notify(SC_ZERO_TIME).** La notification **temporisée** permet le déclenchement de l'événement après un temps spécifié. Ceci se fait par la commande *notify* avec argument : **notify(10, SC_NS).**

Un évènement est souvent associé à un changement d'état dans le processus ou dans le canal. D'un côté, un processus s'occupe de déclencher l'événement avec la méthode **notify(),** et l'autre côté, un processus est sensible à l'événement grâce à la fonction **wait(Evenement).** Une fois notifié, l'événement informe l'ordonnanceur quel est le processus à exécuter.

ANNEXE II

LES BUS STANDARDS

Bus OPB :

Le chronogramme II.1 présente un transfert normal sur le bus OPB. Un maître OPB M1 lit un esclave Sl2. M1 demande le bus OPB en levant le signal de requête (*M1_request*). L'arbitre accorde le bus OPB en levant le signal *OPBM1_Grant* pendant un cycle. Le maître M1 prend contrôle du bus OPB en levant le signal *M1_select*. L'esclave reçoit la requête de lecture, envoie la donnée et l'acquittement dans le même cycle. À la réception de l'acquittement, le maître abaisse le signal *M1_select* libérant le bus.

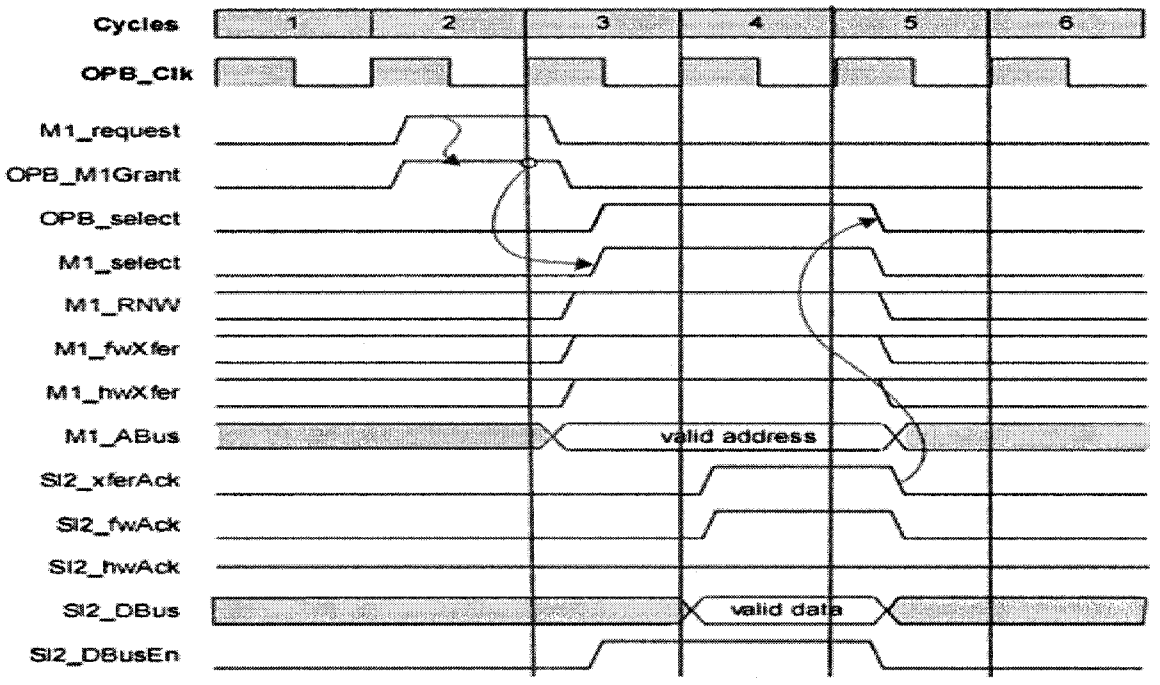

Figure II.1 Transfert simple sur le bus OPB

Le chronogramme II.2 présente un transfert en mode verrouillage (bus lock) sur le

bus OPB. Ce mode permet en transfert de données en continu sans interruption. Dans le chronogramme, les maîtres OPB M1 et M2 lisent le même esclave Sl3. M1 est plus prioritaire que M2. Lorsque M1 obtient le bus, il lève le signal *M1_select* et *M1_buslock* permettant de bloquer le bus. M1 peut alors recevoir les 4 données lues de Sl3 (D0 à D3) à chaque cycle. Lors de la réception de la dernière donnée D3, le *signal M1_buslock* est abaissé permettant un nouveau cycle d'arbitrage (cycle 4). M2 a donc tout de suite le bus au cycle suivant (cycle 5). Ce mécanisme de chevauchement d'arbitrage permet d'éviter un cycle de pénalité d'arbitrage. Le mode verrouillage est essentiel pour les communications Space entre modules pour assurer l'atomicité des messages.

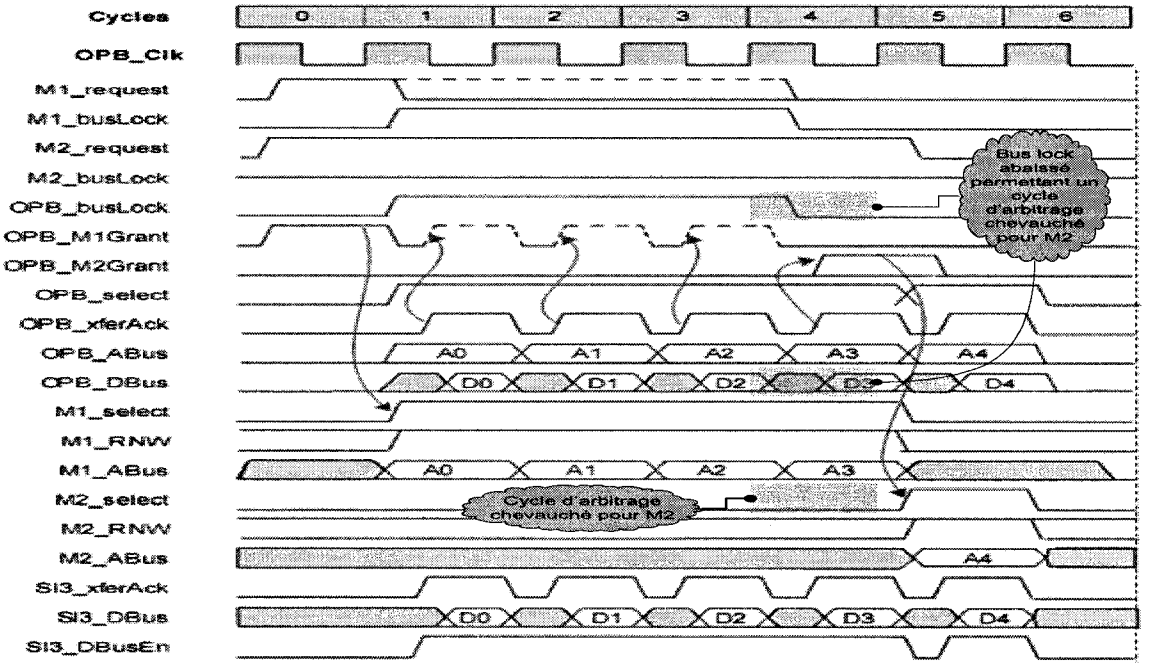

Figure II.2 Mode verrouillage du bus OPB

Comparaison de certains bus standard :

Les tableaux suivants présentent quelques bus standards selon trois catégories : la structure du bus, les types de transferts et l'arbitrage et configuration. Toutes les informations proviennent de la documentation technique des différents bus.

Tableau II.1 Comparaison des caractéristiques de bus stantard - structure

Bus	Hiérarchique	Liens	PàP ou P[1]	Transferts	M-H[2]
AMBA AHB	Oui	Uni	P	Synchrone	Oui
CoreConnect PLB	Oui	Uni	[3]	Synchrone	Oui
CoreFrame MBUS	Oui	Uni	PàP	Synchrone	Oui
Sonics Silicon-Backplane	Oui	Uni	[4]	Synchrone	Oui
STBus	Oui	Uni	P	Synchrone	Oui
Wishbone	Non	Uni/Bi	P	Synchrone	Non

[1] Point à point (PàP) ou partagé (P)
[2] Multi-Horloge
[3] Dans le protocole CoreConnect, le bus de données est partagé et les bus de contrôle forment un anneau
[4] Dans le bus SiliconBackplane, le bus de données est partagé et les signaux de contrôle sont point à point

Tableau II.2 Comparaison des caractéristiques de bus stantard - transferts

Bus	Handshaking	Transactions scindées	Pipeline	Diffusion
AMBA AHB	Oui	Oui	Oui	Non
CoreConnect PLB	Oui	Oui	Oui	Non
CoreFrame MBUS	Non	Non	Non	Non
Sonics Silicon-Backplane	Oui	Oui	Oui	Oui
STBus	Oui	Oui	Oui	Non
Wishbone	Oui	Non	Non	Non

Tableau II.3 Comparaison des caractéristiques de bus stantard - Arbitrage

Bus	Mode	Pipeline	Position	Reconfig. dynamique
AMBA AHB	Défini par l'utilisateur	Oui	Centralisé	Non
CoreConnect PLB	Priorité statique, LRU[1]	Oui	Centralisé	Oui
CoreFrame MBUS	Défini par l'utilisateur	Oui	Centralisé	Oui
Sonics Silicon-Backplane	TDMA + tourniquet	Oui	Distribué	Oui
STBus	7 niveaux (priorité statique, LRU, etc)	Oui	Centralisé	Non
Wishbone	Défini par l'utilisateur	Non	Centralisé	Non

[1] LRU (Least Recently Used): algorithme dynamique où le maître le plus prioritaire est celui qui a eu le bus le moins souvent

ANNEXE III

LES COMPOSANTS DE SPACE

Cette annexe présente un aperçu des composants de Space au niveau TF et BCA. Sur chaque schéma (III.1 et III.2), la représentation graphique SystemC est utilisé pour les ports et les interfaces. Il apparaît aussi un W et un M entouré. Le W indique la présence de **wait()** dans le composant. Cela peut être une attente sur un temps (*wait(10, SC_NS)*), sur un front d'horloge (*wait(clock.pos_egde())*) ou sur un évènement. (*wait(e1)*). La présence d'un *wait()* n'indique pas forcément un processus SystemC dans le composant. Cela signifie que par un appel à une fonction de communication (ModuleRead, ModuleWrite, DeviceRead, DeviceWrite), le processus d'un module utilisateur passera par un *wait()* du à l'imbrication d'appel de fonction à travers les ports/interfaces. Le M signifie Monitoring, i.e. le composant peut être examiné par le monitoring pour recueillir des statistiques de communication et/ou de calcul qui sont analysées en post-simulation.

Niveau TF :

- Module utilisateur : contient le code correspondant à une fonction de l'application. Il communique avec d'autres modules utilisateurs ou des périphériques par les fonctions de communication fournies par Space.
- Module adaptateur : permet de relier le module utilisateur au canal TF. Il forme l'adresse de destination à partir de l'ID du destinataire puis envoie la transaction sur le canal TF. Il reçoit les transactions des autres modules utilisateurs et les stockent dans une zone tampon (*FIFO*).
- Canal TF : possède un arbitre qui décide quel module utilisateur obtient le canal puis achemine la transaction à sa destination.

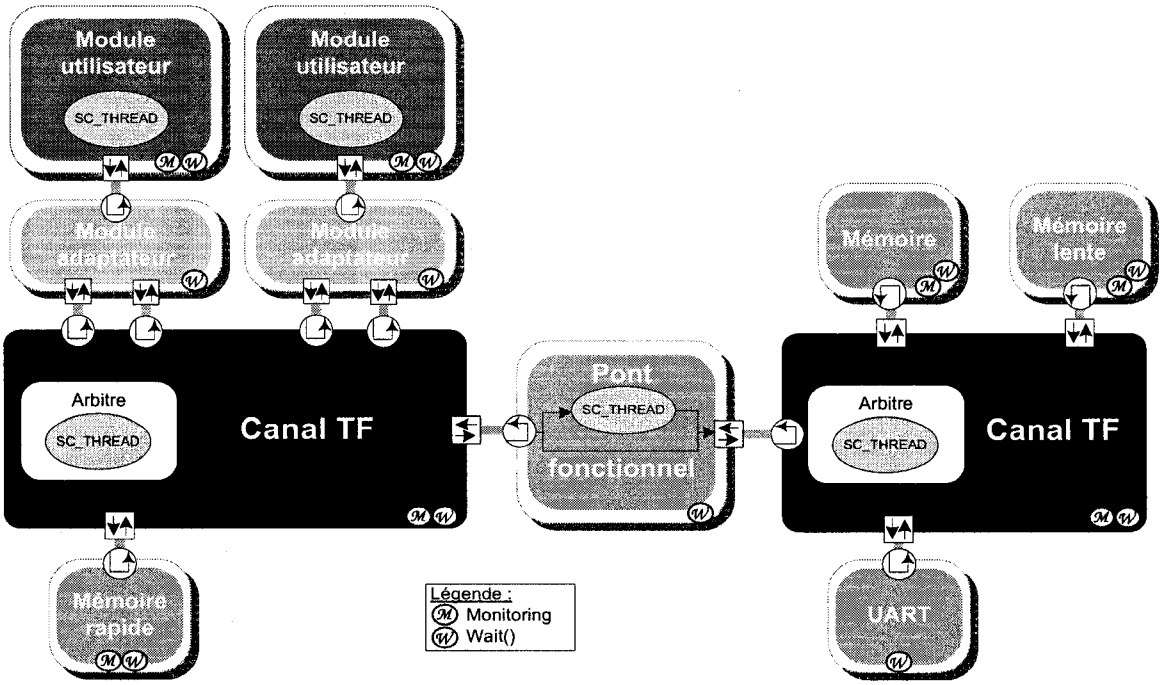

Figure III.1 Composants au niveau TF

- Pont fonctionnel : relie deux canaux TF. Il est bidirectionnel et peut stocker des transactions ou pas.

- Mémoire : périphérique accédé par les modules utilisateurs et pouvant simuler n'importe quel type de latence d'accès.

- UART : périphérique permettant l'émulation d'une communication série via des *sockets* ou une vraie communication par le port série de la machine hôte en utilisant l'API de Windows.

Niveau BCA :

- Module utilisateur : contient le code correspondant à une fonction de l'application. Il communique avec d'autres modules utilisateurs ou des périphériques par les fonctions de communication fournies par Space. Il peut être en matériel ou en logiciel (s'exécutant sur un processeur).

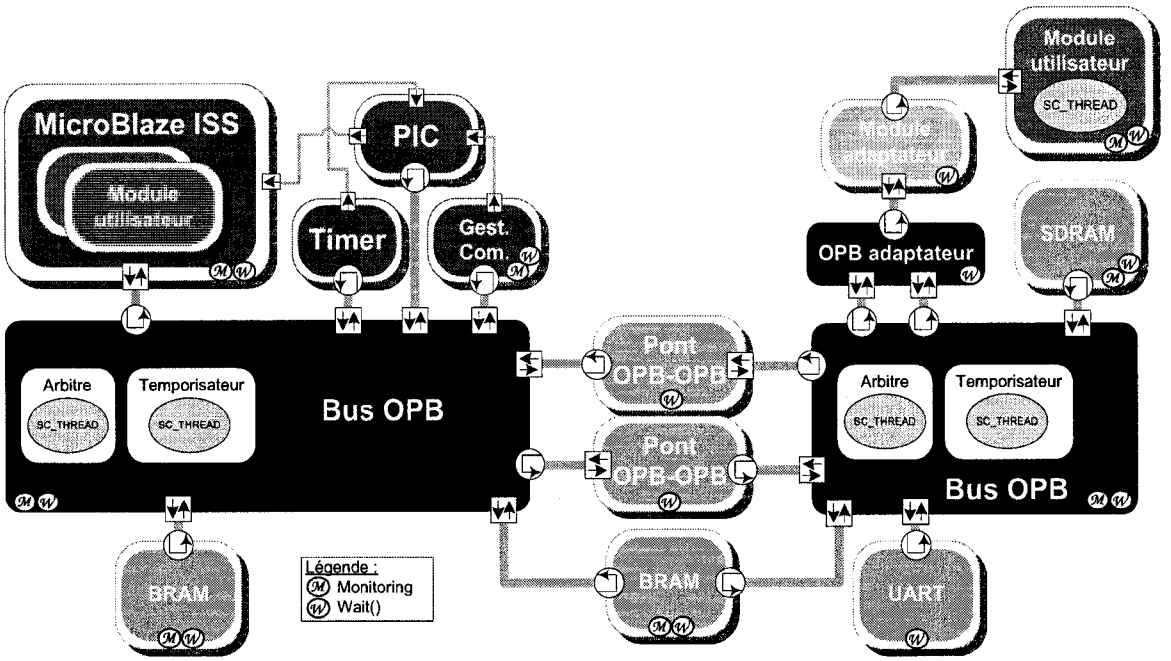

Figure III.2 Composants au niveau BCA

- Module adaptateur : permet de relier le module utilisateur au bus OPB. Il fait le lien entre le protocole de communication propre à Space et le protocole de bus utilisé, ici celui du OPB. Il permet l'envoi des transactions et reçoit celles des autres modules utilisateurs en les stockant dans une zone tampon (*FIFO*).

- OPB adaptateur : transforme la transaction au format Space en transaction répondant au protocole OPB. Il possède une interface maître qui s'occupe d'envoyer les requêtes d'écriture ou de lecture et une interface esclave pour réceptionner les transactions et les transférer au module adaptateur. Il s'occupe de découper une transaction en plusieurs transferts de 32 bits au besoin.

- Bus OPB : possède un arbitre qui décide quel module utilisateur obtient le bus puis achemine la requête à sa destination. Il possède aussi un temporisateur interne pour limiter les accès sur le bus à 16 cycles.

- Pont OPB-OPB : relie deux bus OPB. Il est bidirectionnel et ne stocke pas

transactions. Les bus OPB doivent être de même taille et fonctionner à la même fréquence.

- Mémoire : périphérique accédé par les modules utilisateurs et pouvant simuler n'importe quel type de latence d'accès. La BRAM est une mémoire très rapide simple ou double port sur la puce. La SDRAM est une mémoire rapide hors de la puce.

- UART : périphérique permettant l'émulation d'une communication série via des *sockets* ou une vraie communication par le port série de la machine hôte en utilisant l'API de Windows.

- Microblaze ISS : simule le jeu d'instructions du processeur logiciel, le MicroBlaze de Xilinx. Il peut contenir un système d'exploitation temps réel (RTOS) comme MicroC de Micrium ou Unity qui prend en charge les modules utilisateurs en tant que tâche.

- Timer : minuterie qui permet de cadencer le RTOS du MicroBlaze.

- PIC (*Pin Interrupt Controller*) : gestionnaire d'interruption qui permet d'avertir le MicroBlaze d'un évènement extérieur, i.e. communication avec un périphérique (e.g timer, gestionnaire de communication).

- Gest. Com. : gestionnaire de communication qui permet à un module utilisateur matériel de communiquer avec un module utilisateur logiciel. Les transactions sont stockées dans une zone tampon de lecture. Par l'intermédiaire du PIC, il avertit le processeur qui récupère la transaction stockée en plusieurs accès sur le bus OPB.

ANNEXE IV

LATENCES DES BUS STANDARDS POUR LE CANAL TF

Tableau IV.1 Latences des bus standards pour le canal TF

Bus[1]	Arbitrage	Décodage adresse	Transfert de données	Acquittement
PLB	1 cycle[2]	0, 1 ou 2 cycles[2]	Dépend de la taille de la donnée et du bus	1 cycle
OPB	1 ou 2 cycles	-	Dépend de la taille de la donnée et du bus	1 cycle
AHB	1 cycle	1 cycle	Dépend de la taille de la donnée et du bus[5]	(1 cycle)[5]
ASH	1 cycle	0 ou 1 cycle[3]	Dépend de la taille de la donnée et du bus[6]	(1 cycle)[6]
APB	-	1 cycle[4]	Dépend de la taille de la donnée et du bus	-

[1] le bus APB de Amba est utilisable pour le bus TF grâce au paramètre NO_ARBITRATION. Le bus APB sert à connecter des périphériques. Il ne supporte qu'un maître et plusieurs esclaves car il ne possède pas d'arbitre.

[2] sur le bus PLB, l'adresse se compose de 3 phases : **requête, transfert** et **acquittement. Requête** correspond à l'arbitrage, **transfert** au transfert de l'adresse sur le bus et **acquittement** à l'acquittement de l'adresse par l'esclave (nota bene : dans le bus PLB, il y a 2 acquittements, 1

pour l'adresse, l'autre pour la donnée). La phase d'adresse peut se faire en 1 cycle (les 3 phases se font toutes dans le même cycle), en 2 cycles (cycle 1 : requête, cycle 2 : transfert, ack) ou en 3 cycles (toutes les phases prennent 1 cycle). Ce fonctionnement est représenté dans le tableau par l'arbitrage en 1 cycle et le décodage d'adresse en 0, 1 ou 2 cycles.

[3] sur le bus ASH, il y a forcément un cycle de décodage appelé *decode cycle.* Il ajoute automatiquement un *wait state* pour permettre au décodeur de décoder une adresse stable. Il peut y avoir plus d'un *wait state.* MAIS pour les systèmes à fréquence basse, ce cycle de décodage peut être enlevé.

[4] dans le bus APB, il y a une machine à état qui gère le fonctionnement du bus. Cette FSM est composée de 3 états : IDLE, SETUP et ENABLE. Les états SETUP et ENABLE sont les plus intéressants. SETUP correspond au décodage d'adresse et ENABLE au transfert de la donnée vers le périphérique approprié. SETUP prend 1 cycle.

[5] 1 cycle pour une réponse OKAY simple, 2 cycles pour un RETRY, ERROR ou SPLIT. Dans le cas OKAY, le cycle d'acquittement se fait en même temps que le transfert de la donnée entre le maître et l'esclave. Pour les autres cas, on considère le cycle entre parenthèses pour l'acquittement.

[6] après le cycle de décodage, le transfert consiste à l'envoi de données entre le maître et l'esclave et à la réponse de l'esclave à la requête du maître. D'après la documentation, ces 2 opérations se font dans la même phase. La réponse de l'esclave a lieu à la fin de la dernière donnée à envoyer (si 1 seul donnée, donnée et réponse dans le même cycle). Dans le cas d'une réponse RETRACT de l'esclave, il faut compter 2 cycles pour la réponse (i.e. l'acquittement). C'est pour cela que dans le tableau, il y a 1 cycle entre parenthèses puisque l'autre cycle de l'acquittement est confondu avec le transfert de la donnée.

ANNEXE V

MODÈLES OPB DE XILINX ET ABSTRAIT

Tableau V.1 Modèles OPB de Xilinx et modèle OPB abstrait

Caractéristiques	**Xilinx**	**Modèle abstrait**
Nombre de maîtres et d'esclaves	16 maîtres 16 esclaves	255 maîtres (modules) et 255 esclaves (périphériques)
Arbitrage	Priorité statique Dynamique (LRU)	FIFO, priorité statique, dynamique (LRU)
Taille des transferts	Illimité	Message Space = entête (4 octets) + données (256 octets maximum selon le champ taille de l'entête)
Taille des bus	Adresse: 32 bits Données: 32 à 64 bits	Adresse: 32 bits Données: 32 bits
Suppression du *time-out*	Au choix de l'esclave	Toujours activé, sauf pour le pont OPB-OPB pour gérer les interblocages
Mode verrouillage	Oui	Oui. Sert à l'atomicité des transactions Space.
Mode *Retry*	Au choix de l'esclave	Non géré. Ce cas ne se produit jamais.
Mode séquentiel	Oui	Oui. Permet un mode pseudo-rafale améliorant la bande passante
Dynamic bus sizing	Oui	Non
Transferts DMA	Oui	Non
Mode parcage	Oui	Non
Cycle d'arbitrage chevauché	Oui	Non

ANNEXE VI

RÉSULTATS

Comparaison des estimations de simulation au niveau TF et BCA :

Tableau VI.1 Abréviations

Abréviation	Signification
Tp	Temps physique (s)
Ch	Nombre de cycles d'horloge
Dt	Délai moyen par transaction (cycles)
Ps	Performance de simulation (cycles/s)
N	Nombre d'itérations de chaque séquence
$T_{finSimu}$ et $T_{debutSimu}$	Temps obtenu par sc_simulation_time()
T_{fin} et T_{debut}	Nombre de ticks du système obtenu par clock()

Les formules suivantes permettent le calcul de certaines métriques :

$$Dt = \frac{t_{finSimu} - t_{debutSimu}}{T_H} * \frac{1}{N}$$

$$Ps = \frac{t_{finSimu} - t_{debutSimu}}{t_{fin} - t_{debut}} * CLOCKS\_PER\_SECOND$$

Les tableaux VI.2 et VI.3 présentent diverses métriques pour le banc de test Producteur-Consommateur respectivement pour une architecture à un bus et une architecture à deux bus. Le tableau VI.4 présente diverses métriques pour l'application JPEG pour une architecture à un bus et à deux bus.

Tableau VI.2 Comparaison TF-BCA (un bus) - Producteur-Consommateur

Séquence	Bus	Tp (s)	Ch	Dt (cycles)	Ps (cycles/s)
1	TF	200	100 000 030	10	498 600
1	OPB	431	100 000 030	10	231 570
2	TF	668	420 000 030	42	628 010
2	OPB	1 770	420 000 030	42	237 170
3	TF	81	50 647 198	5	624 670
3	OPB	152	50 958 494	5	334 910
4	TF	145	75 327 710	15	516 622
4	OPB	296	75 491 550	15	254 900

Tableau VI.3 Comparaison TF-BCA (deux bus) - Producteur-Consommateur

Séquence	Bus	Tp (s)	Ch	Dt (cycles)	Ps (cycles/s)
1	TF	373	230 000 030	23	615 280
1	OPB	810	230 000 030	23	283 630
2	TF	782	1 030 000 030	103	1 316 600
2	OPB	3 386	1 030 000 030	103	304 196
3	TF	155	101 597 470	10	652 510
3	OPB	314	101 916 958	10	324 220
4	TF	280	165 819 230	33	590 470
4	OPB	571	165 983 070	33	290 570

Tableau VI.4 Comparaison TF-BCA (Application JPEG)

Nombre de bus	1		2	
Type de bus	TF	OPB	TF	OPB
Temps physique (s)	24	18	57	52
Nombre de cycles d'horloge	11 857 906	8 656 714	14 333 171	9 610 738
Performance de simulation (cycles/s)	494 079	480 929	251 459	184 822